同济大学 TONGJI UNIVERSITY

生物信息学系 DEPARTMENT OF BIOINFORMATICS

Section 08

Managing Complexity with Classes Debugging

Yong Zhang, Ph. D

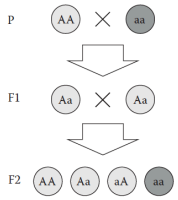
School of Life Science and Technology
Tongji University

Apr 18, 2019

yzhang@tongji.edu.cn

Story: Mendelian inheritance

- To simulate hybridization experiments on peas like Gregor Mendel did.
  - The dominant allele (yellow color) is represented by “G”.
  - The recessive allele (green color) is represented by “g”.



Example Python Session

```
class Pea:

    def __init__(self, genotype):
        self.genotype = genotype

    def get_phenotype(self):
        if "G" in self.genotype:
            return "yellow"
        else:
            return "green"

    def create_offspring(self, other):
        offspring = []
        new_genotype = ""
        for haplo1 in self.genotype:
            for haplo2 in other.genotype:
                new_genotype = haplo1 + haplo2
                offspring.append(Pea(new_genotype))
        return offspring

    def __repr__(self):
        return self.get_phenotype() + ' [%s]' % self.genotype
```

Example Python Session

```
>>> from pea import Pea
>>> yellow = Pea("GG")
>>> green = Pea("gg")
>>> f1 = yellow.create_offspring(green)
>>> f2 = f1[0].create_offspring(f1[1])
>>> print f1
[yellow [Gg], yellow [Gg], yellow [Gg], yellow [Gg]]
>>> print f2
[yellow [GG], yellow [Gg], yellow [gG], green [gg]]
```

Classes are used to create instances

- A class is an abstract representation of real or imaginary things.
- The class defines how the things it represents will behave, but it does not contain any specific data.
  - The *Pea* class is the “idea” of all peas in the platonic sense (all peas have a genotype, require other peas to create next generations of peas, etc).
- Concrete data are in objects generated from a class that are called *instances*.
  - A concrete pea in Python is an instance (e.g. the "*yellow = Pea("GG")*" pea characterized by the "GG" genotype).

Defining a class

- Classes are always defined by the *class* keyword, followed by the name of the class.
- The line starting with class ends with a colon.

```
class Pea:
```

- The entire following indented code block belongs to that class.

The constructor __init__()

- The constructor *__init__()* is a special function that defines what kinds of data your class should contain.

```
def __init__(self, genotype):
    self.genotype = genotype
```

  - It means that a pea has the genotype specified by the second argument of the constructor.
- Internal variables are called *attributes*.
  - The attributes work much like normal variables, only that each variable is preceded by *self*.

How to create instances

- To create instances, you call the class similarly as you do a function, passing as arguments all the parameters required by the constructor (you do not have to explicitly provide a value for the *self* argument).

```
>>> yellow = Pea("GG")
>>> green = Pea("gg")
```

  - These commands create two pea instances, each having a different genotype.
  - Each instance is stored in its own variable: they are really different peas, but both can be used in the same way; they have the same attributes and methods of the Pea class.

Classes contain data in the form of attributes

- Data inside instances are stored in *attributes*. You can access them by using the dot syntax, e.g., *yellow.genotype*.
- Attributes can be changed dynamically like variables.

```
yellow.genotype = 'Gg'
```

Classes contain methods

- The functions within a class are called methods. Methods are used to work with the information within a class.

```
def get_phenotype(self):
    if "G" in self.genotype:
        return "yellow"
    else:
        return "green"
```

- Methods can have default values and optional parameters in the same way that functions do. One class can have methods that analyze, edit, or format the data.

The parameter *self*

- The main difference between a normal function and a method is that methods contain the *self* parameter, which contains the instance for which the method was called.
- With *self* you can access all attributes of a class (e.g., *self.genotype*) and methods (e.g., *self.get_phenotype()*).
- The *self* parameter is automatically passed to the method. Therefore, you always call a method with one fewer parameter than there is in the method definition.

```
>>> yellow = Pea('Gg')
>>> print yellow.get_phenotype()
yellow
```

The *__repr__* method

- You can print objects in a more meaningful way by adding a special method called *__repr__()* to a class:

```
def __repr__(self):
    return self.get_phenotype() + ' [%s]' % self.genotype
```

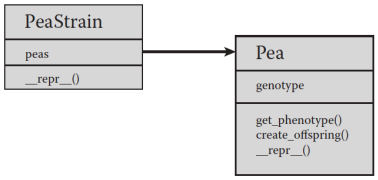
  - Whenever you print a *Pea* instance, *__repr__()* will be called automatically.
  - The *__repr__()* method is also called when you print a list containing *Pea* instances or convert an instance to a string using str().
  - Generally *__repr__()* helps to nicely format your data (i.e. you don't have to include all information in the returned string).

Using classes helps to master complex tasks

- When you define a class, you only need to decide which attributes and methods the class should contain.
- The purpose of using class is to make your program easier to read and understand, not to make it more complicated.
- A good class helps your code become more self-explanatory and reusable.

Examples

- Combining two classes.
  - To have a *PeaStrain* class that manages a group of peas.



Examples

```
from pea import Pea

class PeaStrain:
    def __init__(self, peas):
        self.peas = peas

    def __repr__(self):
        return 'strain with %i peas'%(len(self.peas))

yellow = Pea('GG')
green = Pea('gg')
strain = PeaStrain([yellow, green])
print strain
```

Examples

- Creating subclasses.
  - The attributes and methods of a class can be inherited from other classes. The class inherited from is called base class or parent class, and the inheriting class is called derived class or subclass.
  - To allow peas to contain comments, you could define a class *CommentedPea* inheriting from the *Pea* class.

Examples

```
from pea import Pea

class CommentedPea(Pea):

    def __init__(self, genotype, comment):
        Pea.__init__(self, genotype)
        self.comment = comment

    def __repr__(self):
        return  '%s [%s] (%s)' % (self.get_phenotype(), self.genotype, self.comment)

yellow1 = CommentedPea('GG', 'homozygote')
yellow2 = CommentedPea('Gg', 'heterozygote')
print yellow1
```

Story: When your program does not work

- The broken program is supposed to sort dendritic lengths from a text file into three categories (how many neurons are shorter than 100 µm, how many neurons are longer than 300 µm, and how many neurons are in between). The input text file contains two columns with primary and secondary dendritic lengths:

```
Primary       16.385
Primary       139.907
Primary       441.462
Secondary     29.031
Secondary     40.932
Secondary     202.075
Secondary     142.301
Secondary     346.009
Secondary     300.001
```

Example Python session (with bugs)

```
def evaluate_data(data, lower=100, upper=300):
    """Counts data points in three bins."""
    smaller = 0
    between = 0
    bigger  = 0

    for length in data:
        if length < lower:
            smaller = smaller + 1
        elif lower < length < upper:
            between = between + 1
        elif length > upper:
            bigger = 1
    return smaller, between, bigger

def read_data(filename):
    """Reads neuron lengths from a text file."""
    primary, secondry = [], []

    for line in open(filename):
        category, length = line.split("\t")
        length = float(length)
        if category == "Primary"
            primary.append(length)
        elif category == "Secondary":
            secondary.append(length)
    return primary, secondary
```

Example Python session (with bugs)

```
def write_output(filename, count_pri, count_sec):
    """Writes counted values to a file."""
    output = open(filename,"w")
    output.write("category      <100  100-300   >300\n")
    output.write("Primary  :  %5i   %5i   %5i\n" % count_pri)
    output.write("Secondary:  %5i   %5i   %5i\n" % count_sec)
    output.close()

primary, secondary = read_data('neuron_data.xls')
count_pri = evaluate_data(primary)
count_sec = evaluate_data(secondary)
write_output_file('results.txt' , count_pri,count_sec)
```

Syntax errors

```
  File "program_with_bugs.py", line 23
    if category == "Primary"
                           ^
SyntaxError: invalid syntax
```

- *SyntaxError* means that the Python interpreter did not understand a particular line of code and stopped immediately.
- A syntax error is the programming mistake that is easiest to find.
- Python helps you by giving not only the line number (line 23 in this case) but also a symbol ^ indicating where in the line the problem occurred.

How to tackle a *SyntaxError*

- Check the line before the one highlighted in the syntax error message.
- If there is an *if*, *for*, or *def* statement, is there a colon (*:*) at the end of the line?
- If there is a string starting earlier, is it closed properly?
- If there is a list, dictionary, or tuple stretching over multiple lines, is there a closing bracket?
- Check whether spaces and tabs for indentation are mixed in the code.
- Comment the line or the entire section where the error occurs. Does the syntax error disappear?
- Are you trying to run Python 2.7 code with Python 3.x?

Runtime errors

```
Traceback (most recent call last):
  File "program_without_bugs.py", line 37, in <module>
    primary, secondary = read_data('neuron_data.xls')
  File "program_without_bugs.py", line 20, in read_data
    for line in open(filename):
IOError: [Errno 2] No such file or directory: 'neuron_data.xls'
```

- If there are no syntax errors in your code, Python tries to execute your program line by line. All error messages you see from that point on are called runtime errors.
- A common strategy to do this is to read the error message *from the bottom*:
  - The type of error in the last line (an *IOError* in this case)
  - The line where it occurred in the innermost function (line 20 in this case; line 37 just calls the function that leads to line 20)

IOError

- Your program tried to communicate with an input or output device (a file, directory, or website), but something went wrong.
  - With files the most common reason is that the file or directory name is misspelled.
  - It is also possible that the program could not read or write a given file because you as a user have no permission or the file is already open.
  - With web pages, the reason can be a wrong URL or a problem with the Internet connection.

NameError

```
Traceback (most recent call last):
  File "program_without_bugs.py", line 37, in <module>
    primary, secondary = read_data('neuron_data.txt')
  File "program_without_bugs.py", line 26, in read_data
    secondary.append(length)
NameError: global name 'secondary' is not defined
```

- A *NameError* indicates that the name of a variable, function, or another object is unknown to Python at the moment it is encountered.
- A good diagnostic tool for *NameErrors* is to add the line *print dir()* to your program in the line before the error occurs.

NameError

- Frequent reasons for a *NameError* are the following:
  - A name was not imported, i.e. you forgot to import a variable or function from a different module.
  - A variable has not been initialized. For instance, you have a line:

    counter = counter + 1

    you should initialize that variable somewhere above:

    counter = 0
  - A variable or function name is misspelled.

Handling Exceptions

- Sometimes you can expect where the errors are, for example, your program expects a certain data format that is not always there.
- It would be good if your program could anticipate them and react accordingly instead of you having to start debugging each time a problem occurs.

The *try…except* statement

- The set of statements where you anticipate a runtime error is inserted into an indented block starting with *try*, whereas statements that react to the error are inserted in an indented block starting with *except*.

```
try:
    a = float(raw_input("Insert a number:"))
    print a
except ValueError:
    print "You haven't inserted a number. Please retry."
    raise SystemExit
```

The *try…except* statement

- In the previous example, the corresponding indented block will be entered only in case of *ValueError* exceptions; any other type of exception will not be handled.
- If you want to handle more than one exception, you can add as many *except* blocks as the number of exceptions you want to handle.
- If you do not specify any arguments in the *except* statement, the first exception in the *try* block (no matter what type) will cause the program to execute the *except* block.

The *else* statement

- An *else* statement can be optionally added after a *try…except* block. The set of statements controlled by *else* are executed if no exception has been generated in the *try* block.

```
try:
    filename = raw_input("Insert a filename:")
    in_file = open(filename)
except IOError:
    print "The filename %s has not been found." % filename
    raise SystemExit
else:
    for line in in_file:
        print line
        in_file.close()
```

When there is no error message

- Comparing input and output of your program.
  - Using a small example file for testing.

```
category      <100  100-300   >300
Primary  :      2       2      1   should be 4!
Secondary:     12      11      1
```

When there is no error message

- Adding *print* statements.
  - Adding a *print* statement before and after a possibly erroneous piece of code.

When there is no error message

- Using the Python debugger.
  - To use the Python debugger, you need to insert two lines into your program:

```
import pdb
pdb.set_trace()
```

  - When these lines are reached, Python holds the execution and gives you control of the program in a shell window with a few extra commands available:
    - n: executes the next line
    - s: executes the next line but does not descend into functions.
    - l: shows where in the code the program currently is.
    - c: continues execution normally.

Example Python session (without bugs)

```
def evaluate_data(data, lower=100, upper=300):
    """Counts data points in three bins."""
    smaller = 0
    between = 0
    bigger  = 0

    for length in data:
        if length < lower:
            smaller = smaller + 1
        elif lower < length < upper:
            between = between + 1
        elif length > upper:
            bigger += 1    # error 5
    return smaller, between, bigger

def read_data(filename):
    """Reads neuron lengths from a text file."""
    primary, secondary = [], [] # error 3

    for line in open(filename):
        category, length = line.split("\t")
        length = float(length)
        if category == "Primary":    # error 1
            primary.append(length)
        elif category == "Secondary":
            secondary.append(length)
    return primary, secondary
```

Example Python session (without bugs)

```
def write_output(filename, count_pri, count_sec):
    """Writes counted values to a file."""
    output = open(filename,"w")
    output.write("category      <100  100-300   >300\n")
    output.write("Primary  :  %5i   %5i   %5i\n" % count_pri)
    output.write("Secondary:  %5i   %5i   %5i\n" % count_sec)
    output.close()

primary, secondary = read_data('neuron_data.txt')    # error 2
count_pri = evaluate_data(primary)
count_sec = evaluate_data(secondary)
write_output('results.txt' , count_pri,count_sec)    # error 4
```

Examples

- *ImportError*
  - When you see an *ImportError*, it means that Python tried to import a module but failed.
  - You can check the spelling of the module name.
  - You can verify the directory you started the program from.
  - If you import from a Python library you have installed manually, you can try

    import sys

    print sys.path

Examples

- *ValueError*
  - A *ValueError* occurs when two variables for an operation are incompatible.
  - A good diagnostic tool for *ValueErrors* is to add a *print* statement before the line of the error.

Examples

- *IndexError*
  - An *IndexError* occurs when Python fails to find an element in a list or dictionary.

```
>>> data = [1, 2, 3]
>>> print data[3]
Traceback (most recent call last):
    File "<pyshell#3>", line 1, in <module>
       print data[3]
IndexError: list index out of range
```

Examples

- Writing readable code
  - Generally, code is made more readable by good code modularization, by good organization of a programming project, and by well-formatted code.
  - Variable names are better if they explicitly describe the kinds of data, *seq_length* is better than *number*.
  - Function names should start with a verb and contain one to three words: *read_sequence_file* is easier to read than *read* or *seq_file*.
  - Write comments. As a rule of thumb, use comments as headings for paragraphs in your program, and document lines you find difficult.
  - Avoid the *import* * statement.

Summary

- Managing Your Biological Data with Python
  - Chapter 11. Managing Complexity with Classes
  - Chapter 12. Debugging
- Python codes in https://bitbucket.org/krother/python-for-biologists/src/