Good programming practices

LING 334

Spring 2015

Dr. Bozena Pajak

Good programming practices

- Why are we talking about this?
  - It's very likely you'll keep programming beyond this class (not just in Python)
  - Good programming skills are in high demand
  - It's best to learn good habits early on

Programming (in general)

"There are only two kinds of programming languages: the ones people complain about and the ones nobody uses"

(Bjarne Stroustrup, creator of C++)

Programming (in general)

- Python is just one language (and people complain about it)
- But it's easy to learn and very widely used
- And it teaches you basic programming principles useful for other languages (R, JavaScript, MatLab, …)
- Many programming languages are similar…

Programming (in general)

- Example: a for-loop # prints digits 1-10

```
# Python
for counter in range(1, 11):
      print counter
```

```
# R
for (counter in 1:10) {
      print(counter)
}
```

```
# JavaScript
for (var counter = 1; counter < 11; counter++)
{
      console.log(counter);
}
```

Programming (in general)

- Example: an if-else statement

```
# Python
if keyPress == ‘y’:
        response = ‘yes’
elif keyPress == ‘n’:
        response = ‘no’
else:
        response = ‘other’
```

```
# JavaScript
if (keyPress === 'y’)
{
        response = 'yes’;
}
else if (keyPress === 'n’)
{
        response = 'no’;
}
else
{
        response = 'other’;
}
```

```
# R
if (keyPress == 'y') {
        response <- 'yes'
} else if (keyPress == 'n') {
        response <- 'no'
} else {
        response <- 'other'
}
```

Programming (in general)

- Once you learn one programming language:
  - it's relatively easy to learn other languages because you already know the basic concepts and principles of programming
  - (although of course it depends on the language, and it doesn't mean no effort is required)

Programming (in general)

- What's useful regardless of language is *good programming practices*

Good programming practices

1. First, think high-level
2. Write programs for people, not computers
3. Let the computer do the work
4. Make incremental changes
5. Don't repeat yourself (or others)
6. Plan for mistakes
7. Optimize software only after it works correctly
8. Document design and purpose, not mechanics
9. Collaborate
10. Ensure reproducibility

(largely based on Wilson et al., 2014)

1. First, think high-level

- You want to ensure that the program will do what you want it to do
- and that it will do it in the best way

1. First, think high-level

- **Start with pseudocode**
  - before writing any actual code, think what you want your program to do
  - it's useful to make an outline using pseudocode (in plain words)

1. First, think high-level

```
""" this program tests whether an integer is odd or
even """

get an integer
test whether the integer is odd or even
output the answer
```

1. First, think high-level

- **Come up with specific *algorithms***
  - *algorithm*: the approach or method used to solve a problem

1. First, think high-level

```
""" this program tests whether an integer is odd or
even """

get an integer
test whether the integer is odd or even ??
output the answer
```

1. First, think high-level

```
""" this program tests whether an integer is odd or
even """

get an integer

test whether the integer is odd or even
    divide the integer by 2
    check the remainder of the division
    if the remainder is 0, the integer is even
    otherwise, the integer is odd

output the answer
```

1. First, think high-level

- Then, you can implement your program in a specific language

1. First, think high-level

```
""" this program tests whether an integer is odd or
even """

# get an integer
user_num = int(raw_input("Hi! Enter an integer: "))

# test whether the integer is odd or even, and
output the answer
remainder = user_num % 2 # uses the modulus operator

if remainder == 0:
    print "This integer is even."
else:
    print "This integer is odd."
```

2. Write programs for people, not computers

- Make your programs easily readable and understandable by others
  - this includes the future-you!!
- This will facilitate program maintenance (when you need to make changes), as well as sharing programs

2. Write programs for people, not computers

- **Break programs into chunks, easily understandable single tasks**
  - aesthetically: spaces, empty lines, tabs

    (tabs are required in Python, but not in all programming languages)
  - conceptually: design programs in modules

    (more about modules later on)

2. Write programs for people, not computers

this one is hard to read

```
""" this program tests whether an integer is odd or
even """
# get an integer
user_num=int(raw_input("Hi! Enter an integer: "))
# test whether the integer is odd or even, and
output the answer
remainder=user_num%2 # uses the modulus operator
if remainder==0:
    print "This integer is even."
else:
    print "This integer is odd."
```

2. Write programs for people, not computers

✔ this one is more readable: it has spaces and empty lines

```
""" this program tests whether an integer is odd or
even """

# get an integer
user_num = int(raw_input("Hi! Enter an integer: "))

# test whether the integer is odd or even, and
output the answer
remainder = user_num % 2 # uses the modulus operator

if remainder == 0:
    print "This integer is even."
else:
    print "This integer is odd."
```

2. Write programs for people, not computers

- **Make names consistent, distinctive, and meaningful**
  - don't use non-descriptive names like *a* or *foo* (unless it's a counter/index variable)
  - don't use names that are very similar like results and results1

2. Write programs for people, not computers

variable names are meaningless

```
""" this program tests whether an integer is odd or
even """

# get an integer
fred = int(raw_input("Hi! Enter an integer: "))

# test whether the integer is odd or even, and
output the answer
foo = fred % 2 # uses the modulus operator

if foo == 0:
    print "This integer is even."
else:
    print "This integer is odd."
```

2. Write programs for people, not computers

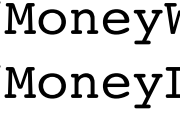
✔ here, the variable names are easier to interpret

```
""" this program tests whether an integer is odd or
even """

# get an integer
user_num = int(raw_input("Hi! Enter an integer: "))

# test whether the integer is odd or even, and
output the answer
remainder = user_num % 2 # uses the modulus operator

if remainder == 0:
    print "This integer is even."
else:
    print "This integer is odd."
```

2. Write programs for people, not computers

- **Make names consistent, distinctive, and meaningful**
  - don't make names *too* descriptive

```
theAmountOfMoneyWeMadeThisYear =
theAmountOfMoneyLeftAtTheEndOfTheYear -
theAmountOfMoneyAtTheStartOfTheYear
```



```
moneyMadeThisYear = moneyAtEnd - moneyAtStart
```

2. Write programs for people, not computers

- **Bottomline about naming variables:**
    - don’t be lazy! put thought into your variable names

2. Write programs for people, not computers

- **Make code style and formatting consistent**
  - it's easier to read if you pick one style (e.g., *MyFunction* or *my_function*)

3. Let the computer do the work

- You want to avoid silly mistakes and save time

3. Let the computer do the work

- **Don't repeat commands manually**
  - write script files and save them
  - build separate smaller programs that are linked together so that a single command can regenerate everything

    ← makefile

    - Makefile resources:
      - » software-carpentry.org/v4/make/
      - » www.slideshare.net/giovanni/makefiles-bioinfo
      - » www.gnu.org/software/make/manual/make.html

3. Let the computer do the work

- **Write functions**
  - if you find yourself writing a similar piece of code multiple times, write a *function* instead
  - "a function is a block of organized, reusable code that is used to perform a single, related action" (tutorialspoint.com/python)
  - Python has many built-in functions: e.g., print()
  - but you can also define your own functions

3. Let the computer do the work

- **Write functions**

```
def functionname(parameters):
        """ function description """
        code
        code
        code
```

3. Let the computer do the work

- **Write functions**

```
def countsToProbs(countsVector):
        """ This function takes as input a vector of counts,
        transforms them into probabilities, and outputs
        vector of probabilities. """

        vectorSum = sum(countsVector)
        probsVector = []
        for count in countsVector:
                probsVector.append(float(count)/vectorSum)
        return(probsVector)
```

1.my_function.py

4. Make incremental changes

- The goal is to avoid unintended functionality, and make debugging efficient

4. Make incremental changes

- **Work in small steps**
  - don't write the whole program from beginning to end, and only run it once you're done
    - it's likely not to work!
    - and then it's hard to find the bug

4. Make incremental changes

- **Work in small steps**
  - instead, write a small chunk and then test it
  - make sure the chunk is doing what it's supposed to
    - to debug use printing and examin your variables from the shell
  - only then move on to the next chunk

2.get_subject_data.py

5. Don’t repeat yourself (or others)

- **Re-use code instead of rewriting it**
  - there’s plenty of code freely available on the web
  - look for packages, modules, functions that might solve your problem before trying to write your own code
    - *module*: a file consisting of Python code; it can define functions, variables, classes (for OOP)
    - *package*: a collection of modules

5. Don’t repeat yourself (or others)

- **Re-use code instead of rewriting it**
  - using modules & packages

```
import os     # os includes miscellaneous operating
              # system interfaces

import os, sys       # sys contains system-level
                     # information, such as the version
                     # of Python you’re running

from sys import path
```

3.get_subject_data_os.py

5. Don't repeat yourself (or others)

- **Modularize code rather than copying and pasting**
  - write your own code with independent modules that contain everything necessary to execute only one aspect of the desired functionality
    - e.g., save your functions in separate files
  - this prevents you from introducing bugs (when you change or fix a piece of code, but don't change it in "code clones")

5. Don't repeat yourself (or others)

- **Modularize code rather than copying and pasting**

```
from my_functions import countsToProbs

# you can call countsToProbs() here
```

  - This works if my_functions.py is in the same directory
  - But you can put my_functions.py in another directory, and add that directory to your Python path; this will let you access it from any other program you write

```
import sys
sys.path.append("/home/me/mypy")
from my_functions import countsToProbs
```

6. Plan for mistakes

- **Remember that everybody makes mistakes**
  - mistakes are inevitable, so be prepared to make them
  - learn efficient debugging techniques

6. Plan for mistakes

- **Work in small steps**
  - as discussed in point 4

6. Plan for mistakes

- **Add assertions to programs to check their operation**
  - an *assertion*: a statement that something holds true at a particular point in a program
  - assertions can be used to ensure that inputs are valid, outputs are consistent, etc.

3.get_survey_data_assert.py

7. Optimize software only after it works correctly

- **Always begin by writing code in the highest-level language possible**

  low vs. high-level languages

  – it's a continuum: from languages "closer to the hardware" (e.g., C) to those with strong abstraction from the details of the computer (e.g., Python, R)

7. Optimize software only after it works correctly

- **Always begin by writing code in the highest-level language possible**

  low vs. high-level languages

  – it's easier and faster to code in high-level languages; the code is more comprehensible

  – but low-level languages are more efficient: programs can be made to run very quickly

7. Optimize software only after it works correctly

- **Always begin by writing code in the highest-level language possible**
  - even if you know you will ultimately need a low-level language, prototype in a high-level language
  - this helps evaluate design decisions quickly
  - it also helps determine which parts are worth optimizing

7. Optimize software only after it works correctly

- **Also apply this rule to code optimization within a language**
  - there might be more efficient ways of writing a particular piece of code
  - but if it's complicated to implement, wait to optimize until you have your program working
  - you might end up changing the design as you work through your program
  - or perhaps this particular optimization will turn out not to be worth the time

8. Document design and purpose, not mechanics

- **Add informative comments**
  - Comments are very important!
  - They help you not waste time on rethinking what a particular piece of code was supposed to do

8. Document design and purpose, not mechanics

- **Add informative comments**
  - at the beginning of any program, add a bigger comment with your name, date, program description

```
##################
# Bozena Pajak #
# 5/4/2015     #
##################

""" This program combines individual subject files
into one file. """
```

8. Document design and purpose, not mechanics

- **Add informative comments**
  - at the beginning of each function, describe what it does and its inputs and outputs

```
def countsToProbs(countsVector):
        """ This function takes as input a vector of counts,
        transforms them into probabilities, and outputs
        vector of probabilities. """

        vectorSum = sum(countsVector)
        probsVector = []
        for count in countsVector:
                probsVector.append(float(count)/vectorSum)
        return(probsVector)
```

8. Document design and purpose, not mechanics

- **Add informative comments**
  - add other comments about the logic of your code throughout

```
# store survey info in a dictionary with subject
# number as a key and the rest as the value
surveyDict = {}

for line in rf:
    line = line.strip().split('\t')
    subjNum = line[0]
    subjInfo = ','.join(line[1:])
    surveyDict[subjNum] = subjInfo
```

8. Document design and purpose, not mechanics

- **Add informative comments**
  - don't add comments just for the sake of commenting: some comments are useless, and clog the code
  - (although it's fine to comment more when you're learning to program)

```
# store survey info in a dictionary with subject
# number as a key and the rest as the value

# initialize dictionary for survey data
surveyDict = {}

# loop through each line of the survey file
for line in rf:

    # strip end of line char, and split the line into a list
    # by tab
    line = line.strip().split('\t')

    # save subject number
    subjNum = line[0]

    # join all list items (except subject number) by tab
    subjInfo = ','.join(line[1:])

    # save info in the dictionary
    surveyDict[subjNum] = subjInfo
```

some of the commented code is self-evident
some comments only describe the mechanics

8. Document design and purpose, not mechanics

- **Important rules for commenting**
  - use plain language
  - add comments as you go
    - nobody likes documenting code: you are unlikely to go back and add comments later
    - it's easier to do when the logic is fresh in your mind
    - it can help you structure the code better: if a substantial description of a piece of code is needed, consider reorganizing the code so that it's more straightforward
  - get in the habit of commenting early on!!

9. Collaborate

- **Have your code reviewed by somebody else**
  - reviews of code can eliminate bugs and improve readability
  - this is also a good way to spread knowledge and good practices around a team
    - code reviews help ensure that critical knowledge isn't lost when the programmer leaves

9. Collaborate

- **Have your code reviewed by somebody else**
  - extreme version: *pair programming*
    - many programmers find it intrusive
    - but it can be helpful when tackling particularly tricky problems

10. Ensure reproducibility

- **Document everything**
  - people largely overestimate how much they will remember
  - keep read-me files that explain what you did
  - your records should be understandable by another person
  - keep all the files organized, with meaningful names

10. Ensure reproducibility

- **Use a version control system (VCS):**
  e.g., Git, Mercurial, Subversion
  - VCS stores a snapshot of a project's files in a *repository*
  - users can modify their working copy, and then *commit* changes to the repository

10. Ensure reproducibility

- **Use a version control system (VCS):**
  e.g., Git, Mercurial, Subversion
  - good for collaborations
    - VCS resolves any conflicts before accepting changes
    - it stores the entire history of the files, allowing different versions to be retrieved and compared
    - it also stores metadata: users' comments on what was changed

Good programming practices

1. First, think high-level
2. Write programs for people, not computers
3. Let the computer do the work
4. Make incremental changes
5. Don't repeat yourself (or others)
6. Plan for mistakes
7. Optimize software only after it works correctly
8. Document design and purpose, not mechanics
9. Collaborate
10. Ensure reproducibility

(largely based on Wilson et al., 2014)

Python resources

- Python tutorials
  - www.tutorialspoint.com/python
  - docs.python.org/3/tutorial
  - (plenty more on the web)
- Python documentation
  - docs.python.org
- Software Carpentry: general computing
  (mission: teaching scientists basic computing skills)
  - software-carpentry.org