Program Style and Design

Overview

- An introduction to *program style*
- Functional decomposition
- Designing programs
  - What is design and how do we do it?
- Back to style again
  - Our style guidelines
  - *pylint*

What is style?

- *Style:* "The way in which something is said, done, expressed, or performed"
- In programming, *style* is concerned with *readability* and *maintainability*
- In MSE 800, the focus is on *correctness* and *style*
    - *Efficiency* takes a back seat (or *should* do)
        - Worry about it only when it has **proven** to be a problem
        - Slow right answers are better than fast wrong ones
    - Even "throw-away" code needs to be readable and correct
        - Well-written code is easier to debug

Functional decomposition

Functional decomposition 1

- Recap: key aspects of style
- The what and why of functional decomposition
  - Why functions are good for you and how to get them.

Key aspects of program style

- Is the code readable and maintainable?

1. Good identifiers?
2. Good layout?
    - White space; line length <= 80
3. Good breakdown into functions?
    - Clarity
    - Avoidance of repetition

functional decomposition. We'll look at this first.

4. Has the complexity been kept under control?
    - No more than 4 levels of indentation (e.g. function, 2 loops and an if)
    - Functions at most 40 lines in length, including comments
5. Docstrings for the program as a whole and all functions?

Functional Decomposition

- Why we use functions
- The two sorts of functions
- “Discovering” functions
  - Function extraction
  - Top-down design

Two main reasons to write functions

1. To make the code easier to understand
   - Break a large problem into smaller *named* chunks
   - Work on one subproblem at a time

Copy and paste is *evil!*

2. To make code *reusable*
   - In the same module
   - In *other* modules

The two sorts of functions

Procedures

("Write a function that prints …")

- **Don't** return a value to caller
  - No return statement (in Python they implicitly return None)
- **Do** print output (or write files etc)
- Names start with a verb
  - *print_table, compute_results*
- Called as, e.g.,
  - `print_table(10)`
  - `compute_results(data)`

Real functions

("Write a function that returns …")

- **Do** return a value to caller
  - Must have a return statement
- **Don't** print output, write files (usually)
- Names are nouns
  - *standard_error, max_rainfall*
- Called as, e.g.,
  - `error = standard_error(data)`
  - `print(max_rainfall(data))`

Functional decomposition 2

- How functions arise
- Extracting arbitrary code as a function

How do functions arise?

- Functions arise:
  - Deliberately, from top-down design (coming soon)
    - e.g. start with idea of *read_and_process_data*, decompose to two functions *read_data* and *process_data*. Decompose latter to *get_site_info, get_rainfalls, compute_averages*, etc.
  - During *refactoring* when trying to clean up code
    - "Function extraction"
  - By discovery
    - e.g. a particular sequence of statements seems to occur repeatedly. Aha, that's a "thingy" function.
- Decomposing a program or top-level function into smaller functions is *functional decomposition*

Functional extraction: example

Before

```python
def process_data(data):
    # Step 1: Clean data
    cleaned_data = []
    for item in data:
        if isinstance(item, str):
            item = item.strip()
        cleaned_data.append(item)

    # Step 2: Transform data
    transformed_data = []
    for item in cleaned_data:
        if isinstance(item, str):
            item = item.upper()
        transformed_data.append(item)

    # Step 3: Summarize data
    summary = {}
    for item in transformed_data:
        if item in summary:
            summary[item] += 1
        else:
            summary[item] = 1

    return summary

data = [" apple", "banana ", " apple ", "BANANA", "Apple"]
result = process_data(data)
print(result)
```

After

```python
def clean_data(data):
    cleaned_data = []
    for item in data:
        if isinstance(item, str):
            item = item.strip()
        cleaned_data.append(item)
    return cleaned_data

def transform_data(data):
    transformed_data = []
    for item in data:
        if isinstance(item, str):
            item = item.upper()
        transformed_data.append(item)
    return transformed_data

def summarize_data(data):
    summary = {}
    for item in data:
        if item in summary:
            summary[item] += 1
        else:
            summary[item] = 1
    return summary

def process_data(data):
    cleaned_data = clean_data(data)
    transformed_data = transform_data(cleaned_data)
    summary = summarize_data(transformed_data)
    return summary

data = [" apple", "banana ", " apple ", "BANANA", "Apple"]
result = process_data(data)
print(result)
```

Extracting a function

- Any sequence of complete statements can always be pulled out into a separate function by:
  - Identifying all the variables that must be defined already for the statements to execute
    - These are the *input parameters*
  - Identifying all the variables that get defined or altered by the statements and are needed by later code
    - These, collectively, are the *return value* (e.g. a tuple)
- BUT for this to be **sensible**, the function should have a role that can be concisely summarised by the function's name.
  - And the number of input parameters and output values must be *small*.

A bad refactoring

```
def main():
    """Main function. Read file. Count required attributes"""
    filename = input("Enter program filename: ")
    lines = open(filename).readlines()
    num_for_loops = 0
    for line in lines:
        if line.strip().startswith('for '):
            num_for_loops += 1

    print("Program {} contains {} for loops".format(filename,
       num_for_loops))

    num_while_loops = 0
    for line in lines:
        if line.strip().startswith('while '):
            num_while_loops += 1

    print("Program {} contains {} while loops".format(filename,
       num_while_loops))

    num_if_statements = 0
    for line in lines:
        if line.strip().startswith('if '):
            num_if_statements += 1

    # etc
```

Duplicate code!

Overly complex main function!

Lack of abstraction!

Make this a function:

Input values: num_for_loops, lines, filename
Output value: num_while_loops
(NB: "Output" *doesn't* mean "stuff we print")

Functional decomposition 3

A much better refactoring

- Any sequence of complete statements can be made into a function.
- BUT for this to be **sensible**, the function should have a role that can be concisely summarised by the function's name.
  – And the number of input parameters and output values must be *small*.

Much better attempt at refactoring

```
def main():
    '''Main function: prompt for filename, print various statistics'''
    filename = input("Enter program filename: ")
    lines = open(filename).readlines()
    count_fors(lines, filename)
    count_whiles(lines, filename)
    count_ifs(lines, filename)
    count_defs(lines, filename)
```

```
def count_whiles(lines, filename):
    '''Count and print the number of while loops'''
    num_while_loops = 0
    for line in lines:
        if line.strip().startswith('while '):
            num_while_loops += 1

    print("Program {} contains {} while loops".format(
            filename, num_while_loops))
```

```
def count_fors(lines, filename):
    '''Count and print the number of for loops'''
    num_for_loops = 0
    for line in lines:
        if line.strip().startswith('for '):
            num_for_loops += 1

    print("Program {} contains {} for loops".format(
            filename, num_for_loops))
```

```
def count_ifs(lines, filename):
    '''Count and print the number of if statements'''
    num_if_statements = 0
    for line in lines:
        if line.strip().startswith('if '):
            num_if_statements += 1

    print("Program {} contains {} if statements".format(
            filename, num_if_statements))
```

Even better

- But all those functions are almost identical! So *generalise*

```
def main():
    """Main function: prompt for filename, print various statistics"""
    filename = input("Enter program filename: ")
    lines = open(filename).readlines()
    count_and_print('for ', 'for loop', lines, filename)
    count_and_print('while ', 'while loop', lines, filename)
    count_and_print('if ', 'if statement', lines, filename)
    count_and_print('def ', 'function definition', lines, filename)
```

```
def count_and_print(token, statement_type, lines, filename):
    """Count and print the number of occurrences of the given
       start-of-line token in the given list of lines from the given
       file. Print a message showing the count. 'statement_type'
       is the type of statement corresponding to the given token.
    """
    num_statements = 0
    for line in lines:
        if line.strip().startswith(token):
            num_statements += 1

    print("Program {} contains {} {}s".format(
            filename, num_statements, statement_type))
```

Program Design 1

- Black art!
- Common methods:
  - Top-down (task to details)
  - Bottom-up (details to task)
  - Object-oriented (data-driven)

You can mix methods

Top-down design

- **<u>Ideally</u>** we don't have to extract functions from existing code
- We write it right the first time!
  - Write the main function as a call to some to-be-written support functions
  - Write an empty implementation for each of those
  - Then code each support function as calls to sub-functions
  - And so on.
- Called *hierarchical decomposition* and/or *top-down design*.

Top-down design (cont'd)

- State the problem (may be a whole document!)
- Decide "logically" how it should work
  - Hierarchical: start with the big picture and continually refine down to the detail
- Translate the logical program into code
  - Detailed logic into **statements**
  - Related statements into **functions**
  - Related functions into **modules**
- **Refactor** code to improve the design

Design 2

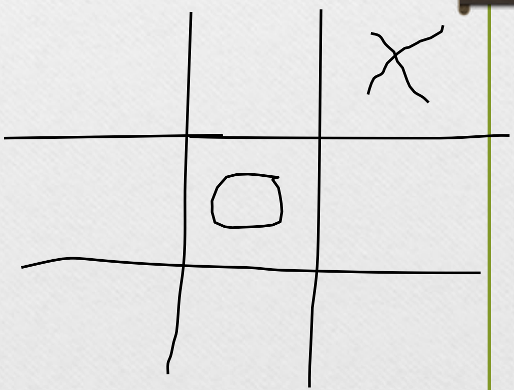
A demo of how the top-down design approach might work for a noughts and crosses ("tic-tac-toe") program.

https://en.wikipedia.org/wiki/Tic-tac-toe

Noughts and Crosses

- Human versus computer. Human first. 3x3 board.
- Top-level pseudocode:
    1. *Make a board*
    2. *Determine if the user wants to be 'O' or 'X'*
    3. *Play the game until a win or a draw*
    4. *State the outcome*
- Or, in top-level Python:

```
def main():
    """Play a game of noughts and crosses"""
    board = new_board()
    human_player = get_O_or_X()
    outcome = play_game(board, human_player)
    print_result(outcome)
```



e.g., Noughts and Crosses (cont.)

- Function:
  Play_the_game_until_a_win_or_a_draw
- Pseudocode refinement...

  while still playing

      display the board

      play a turn for whoever's go it is

      check if there's a win or a draw
- Or, level-2 Python:
-

```
def play_game(board, player):
    """Play until a win or a draw"""
    current_player = player
    while not game_over(board):
        display(board)
        play_one_turn(board, current_player)
        current_player = other_player(current_player)
```

e.g., Noughts and Crosses (cont.)

- Function : Play_one_turn
- Pseudocode unchanged

---

if human_turn

human_move(board)

else

computer_move(board)

- Python level 3 - need to add an extra parameter
-

```
def play_one_turn(board, current_player, human player):
    """Given a board state and the current
      player ('O' or 'X'), play one move"""
    if current_player == human_player:
        make_human_move(current_player, board)
    else:
        make_computer_move(current_player, board)
```

e.g., Noughts and Crosses (cont.)

- Function: make_human_move
- Pseudocode:

while not valid_choice

position = get_a_location()

valid_choice = board_location_is_empty(board, position) if not valid_choice

write error message to user

- Python level 4

```
def make_human_move(board, current_player):
    """Given a board state and the human piece ('O' or 'X')
       ask the player for a location to play in. Repeat until a
       valid response is given."""
    ... You do! It's now straightforward python ...
```

e.g., Noughts and Crosses (cont.)

- Function: make_computer_move.
- Pseudocode:
  - Could be hard.
  - For now: *choose a random (empty) square!*
- Python level 4

```
def make_computer_move(board, current_player):
    """Given a board state and the computer piece ('O' or 'X')
       choose a square for the computer to play in"""
    candidates = empty_squares(board)
    choice = random_in_range(0, len(candidates))
    row, column = candidates[choice]
    board[row][column] = current_player
```

e.g., Noughts and Crosses (cont.)

- Function: game_over
- Pseudocode:

  return true if any player has 3-in-a-row or if board is full

- Python level-3

```
def game_over(board):
    """Given a board state return true iff there's a winner or
       if the game is drawn."""
    return won_game(board) or is_full(board)
```

e.g., Noughts and Crosses (cont.)

- Function: won_game (a.k.a. *any player has 3-in-a-row*)

- Pseudocode:

for each row

  win is true if all items are x or o

for each column

  win is true if all items are x or o

for each diagonal

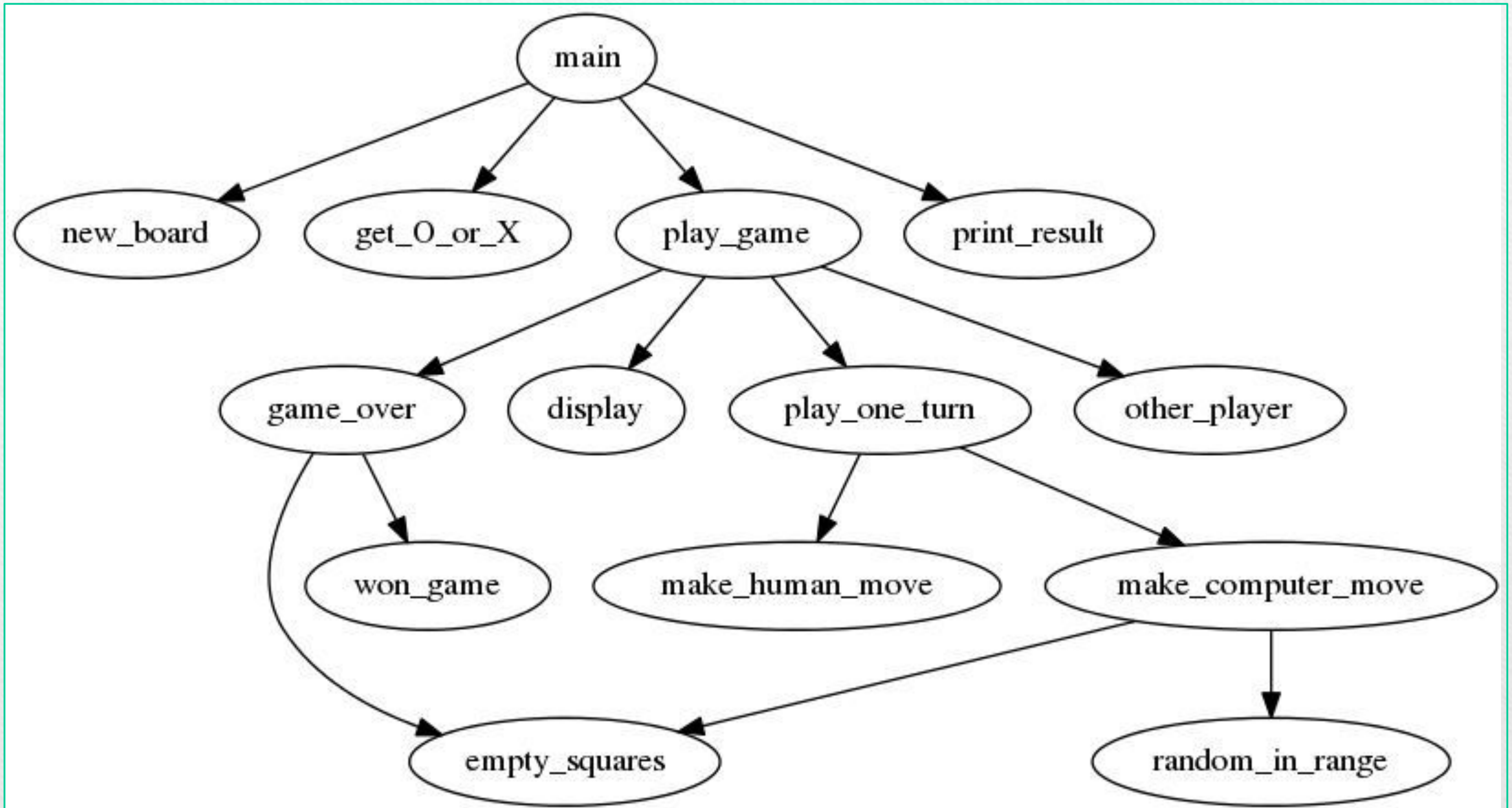
  win is true if all items are x or o

```
def game_over(board):
    """Return true iff there's a winner"""
    ... you do - it's down to basic python again ...
```

And so on …

- Continue until everything is implemented.

Get something like



Style 2

- Back to *Style* again
- *pylint* enforces many basic style rules but it's still possible to run *pylint*-compliant rubbish

https://docs.pylint.org/intro.html#what-is-pylint

Key aspects of program style

- **Is the code generally nice and readable?**
  - Good identifiers?
  - Correct layout?
    - White space; line length <= 80
  - Good breakdown into functions?
    - Clarity
    - Avoidance of repetition
  - Has the complexity been kept under control?
    - No more than 4 levels of indentation (e.g. function, 2 loops and an if)
    - Keep functions to at most 40 lines in length, including comments
  - Explanatory comments at top of module and on all functions?

Green: Checked by *pylint*
Blue: Not checked by *pylint*

Style guidelines

- The style rules according to *pylint*
- "Official" Python standard: PEP-8
  - http://www.python.org/dev/peps/pep-0008/

```
# Correct:

# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

```
# Wrong:

# Arguments on first line forbidden when not using vertical alignment.
foo = long_function_name(var_one, var_two,
    var_three, var_four)
```

Specific rules: variable names

- Use only lower case
  - Separate words by underscore ('_')
  - e.g. use *is_html_compliant* not *is_HTML_compliant*
- Use longer more-meaningful names for 'wide-scope' variables
  - Avoid abbreviations (except common ones like html, cpu, ...)
  - No one-character identifiers except:
    - *i, j, k* for loop control variables used as indices into lists etc
    - *c* for a generic character
    - *s* for a generic string

Function names

- Distinguish between functions that return values (‘real’ functions) and functions that do things (‘procedures’)
  - Name value-returning functions for what they *are*, e.g.
    - *error = standard_error(...)*
      - **Not** *error = compute_standard_error(...)*
    - They generally shouldn’t print or write files
  - Name procedures for what they *do*, e.g.
    - *print_file(..), compute_standard_error(...)*
    - They generally shouldn’t return anything

Program structure

- Break programs into lots of functions
  - – Must be well named, do one thing well
- Don’t use global variables/code (next slide)
- At most 40 lines per function, including comments
- At most 4 levels of indentation (e.g. a function with a loop, a nested loop and a nested if statement)
- Minimise use of *break* and *continue*
- Avoid cryptic code (“clever code isn’t”)
- Don’t rebuild the wheel - use Python’s library functions

Avoid global variables and code

- Don’t use global variables (i.e. variables at the outermost level, with no indentation).
  - But global *constants* are OK

NOT THIS

```
import blah

x = 10
y = int(input('Gimme'))
print('x + y = ', x + y)
```

BUT THIS

```
import blah

def main():
    """A meaningless main func"""
    x = 10
    y = int(input('Gimme'))
    print('x + y = ', x + y)


main()
```

Layout

- Separate functions by 2 or 3 blank lines
- At most 80 characters per line
  - Well, ok, maybe we'll let you go up to 100 *occasionally*
- Don't write multiple statements on one line
- Use white space around binary operators
  - e.g. `n = 25 * (n_chars - 1)`
    not `n=25*(n_chars-1)`

Comments

```
"""
Module Name: data_processing.py
Role: This module contains functions to process and analyze data.
Author: Isabel Wang
Date: 2024-06-02
"""
```

- Every *program* must have a module docstring at the top stating its role, author, date.
- Every *function* must have a docstring at the start specifying what it does.
- Elsewhere, use comments very carefully
  - Good code should be readable without comments

Constants

- Avoid use of hard-coded constant values in the code.
- Instead define them once only at the top of the program.
- Use ALL_CAPS (with underscore to separate words)
  - e.g. DAYS_IN_WEEK, MILES_PER_KILOMETRE
- More maintainable and more readable.
- For example:

Not this

```
while error > 0.00001:
    compute new approximation
```

But this

```
TOLERANCE = 0.00001 # Global
.
.
def somefunction(…) :
    while error > TOLERANCE:
        compute new approximation
```

Exercises

Function Decomposition Exercises_1

Exercise goal: Split the functionality of processing grades into multiple functions, such as calculating the sum, calculating the average, and finding the highest and lowest scores.

```python
def process_grades(grades):
    total = 0
    highest = grades[0]
    lowest = grades[0]
    for grade in grades:
        total += grade
        if grade > highest:
            highest = grade
        if grade < lowest:
            lowest = grade
    average = total / len(grades)
    print(f"Total: {total}")
    print(f"Average: {average}")
    print(f"Highest: {highest}")
    print(f"Lowest: {lowest}")
```

Function Decomposition Exercises_2

Exercise goal: Split the functionality of text analysis into multiple functions, such as segmenting text, counting word occurrences, and finding the most frequent words.

```
def analyze_text(text):
    word_count = {}
    lines = text.split('\n')
    for line in lines:
        words = line.split(' ')
        for word in words:
            if word in word_count:
                word_count[word] += 1
            else:
                word_count[word] = 1
    most_frequent_word = max(word_count, key=word_count.get)
    print(f"Most frequent word: {most_frequent_word}")
    print(f"Word count: {word_count}")
```

Thank You