Programming Principles in Python (CSCI 503/490)

Structural Pattern Matching

Dr. David Koop
CPU-Bound vs. I/O-Bound

CPU Processing

Compute Problem 1  Compute Problem 2

I/O Waiting

Request 1  Request 2  Request 3

CPU Processing

Time
Threading

- Threading addresses the I/O waits by letting separate pieces of a program run at the same time.
- Threads run in the same process.
- Threads share the same memory (and global variables).
- Operating system schedules threads; it can manage when each thread runs, e.g., round-robin scheduling.
- When blocking for I/O, other threads can run.
Python Threading Speed

• If I/O bound, threads work great because time spent waiting can now be used by other threads
• Threads do not run simultaneously in standard Python, i.e. they cannot take advantage of multiple cores
• Use threads when code is I/O bound, otherwise no real speed-up plus some overhead for using threads
Python and the GIL

• Solution for reference counting (used for garbage collection)
• Could add locking to every value/data structure, but with multiple locks comes possible **deadlock**
• Python instead has a Global Interpreter Lock (GIL) that must be acquired to execute any Python code
• This effectively makes Python single-threaded (faster execution)
• Python requires threads to give up GIL after certain amount of time
• Python 3 improved allocation of GIL to threads by not allowing a single CPU-bound thread to hog it
Multiprocessing

- Multiple processes do not need to share the same memory, interact less
- Python makes the difference between processes and threads minimal in most cases
- Big win: can take advantage of multiple cores!
Multiprocessing using concurrent.futures

- import concurrent.futures
  import multiprocessing as mp
  import time

  def dummy(num):
      time.sleep(5)
      return num ** 2

  with concurrent.futures.ProcessPoolExecutor(max_workers=5,
      mp_context=mp.get_context('fork')) as executor:
      results = executor.map(dummy, range(10))

- mp.get_context('fork') changes from 'spawn' used by default in MacOS, works in notebook
asyncio

• Single event loop that controls when each task is run
• Tasks can be ready or waiting
• Tasks are **not interrupted** like they are with threading
  - Task controls when control goes back to the main event loop
  - Either waiting or complete
• Event loop keeps track of whether tasks are ready or waiting
  - Re-checks to see if new tasks are now ready
  - Picks the task that has been waiting the longest
• `async` and `await` keywords
• Requires support from libraries (e.g. `aiohttp`)
When to use threading, asyncio, or multiprocessing?

- If your code has a lot of I/O or Network usage:
  - If there is library support, use asyncio
  - Otherwise, multithreading is your best bet (lower overhead)
- If you have a GUI
  - Multithreading so your UI thread doesn't get locked up
- If your code is CPU bound:
  - You should use multiprocessing (if your machine has multiple cores)
Concurrent Types | Switching Decision | Number of Processors
--- | --- | ---
Pre-emptive multitasking (threading) | The operating system decides when to switch tasks external to Python. | 1
Cooperative multitasking (asyncio) | The tasks decide when to give up control. | 1
Multiprocessing (multiprocessing) | The processes all run at the same time on different processors. | Many
Assignment 7

- Coming soon…
Conditional Logic

• if/elif/else
• What about a switch statement?
• Exists in C++
• What are the advantages of a switch?

• C++ Example:
  ```cpp
  - switch (val) {
    case 1:
      cout << "1st" << endl;
      break;
    case 2:
      cout << "2nd" << endl;
      break;
    default:
      cout << "???
```
Conditional Logic

- if/elif/else
- What about a switch statement?
- Exists in C++
- What are the advantages of a switch?
  - Cleaner and less redundant
  - More efficient than if/elif…

C++ Example:
```cpp
- switch (val) {
  case 1:
    cout << "1st" << endl;
    break;
  case 2:
    cout << "2nd" << endl;
    break;
  default:
    cout << "???"><< endl;
}
```
Python

- Python's if/elif structure is pretty similar structure-wise to a switch statement.
- If you want the "jump" functionality, remember that dictionaries offer efficient lookup and can store functions!
- **Example:**
  ```python
  ops = {
    1: lambda: print('1st'),
    2: lambda: print('2nd')
  }
  ops.get(val, lambda: print('???'))()
  ```
- ... so no great need for a standard switch statement
Match Statement

• But… Python 3.10 added a match statement that can be used like a switch statement

```python
match val:
    case 1:
        print('1st')
    case 2:
        print('2nd')
    case _:
        print('???')
```

• Why?
  - It can do **more** than a switch statement
Structural Pattern Matching

• Besides literal cases, match statements can be used to
  - differentiate structure
  - assign values
  - differentiate class instances

• Example:

```python
match sys.argv:
    case [_, "commit"]:  
        print("Committing")
    case [_, 'add', fname]:
        print("Adding file", fname)
```
Evaluation

- Works similar to if/elif/else logic
- Cases are processed in order
- Once the first case is matched, it's body is executed and no other cases will be matched
- Name bindings (assignments) can be used after the match statement
  - Like standard conditional logic in Python
  - Differs from some other languages where if/then blocks are scoped…
Simple Patterns

- Literal patterns:
  - e.g. 2, "commit", but also True, False, None

- Simple capture pattern:
  - an identifier: fname

- Wildcard: matches anything: _
Sequence Pattern

• A sequence composed of other patterns: ["add", fname]
• Any identifiers are assigned when the structure is matched
• Can allow a match of multiple values using * syntax

```python
- match sys.argv:
  case [_, "commit"]:    
    print("Committing")
  case [_, 'add', *fnames]:
    print("Adding files", fnames)
```
Or Pattern

• May allow multiple patterns to be processed by a single block
• Uses the bar symbol (not the word "or") to connect the patterns
• Example:

```python
match command.split():
    ... # Other cases
    case ["north"] | ["go", "north"]:  
        current_room = current_room.neighbor("north")
    case ["pick", "up", obj] | ["pick", obj, "up"]:  
        ... # Code for picking up the given object
```
Or Pattern

- Problem: Suppose we want to restrict the set of values but don't know which pattern was selected...

```python
match command.split():
    case ["go", ("north" | "south" | "east" | "west")]:
        current_room = current_room.neighbor(...)  # how do I know which direction to go?
```
As Pattern

- Similar to exceptions where we can assign the matched value to an identifier when the patterns are literals
- Can even do this in a more complex pattern:
  
  ```python
  match command.split():
      case ["go", ("north" | "south" | "east" | "west") as d]:
          current_room = current_room.neighbor(d)
  ```
Guards

• In some cases, we want to add additional logic to check the pattern before allowing the block to be executed
• If the guard is not True, **other cases continue to be checked**
• Example: Suppose certain directions are not allowed in a given room:

```python
match command.split():
    case ['go', dir] if dir in current_room.exits:
        current_room = current_room.neighbor(dir)
    case ['go', _]:
        print("Sorry, you can't go that way")
```

[PEP 636]
Matching Types

• You can match a type in a similar manner, but must put parentheses after it
  - case str():

• We can combine this with the as pattern to capture the value
  - case str() as s:

• There is also shorthand to do this (useful in more complex expressions)
  - case str(s):
Class Pattern

- We can also match objects...
- match event:
  case Click() as c:
    print("Click happened", c.x, c.y)

- but the type shortcut **does not work**
- match event:
  case Click(c):
    print("Click happened", c)
  ...

- TypeError: Click() accepts 0 positional sub-patterns (1 given)
Class Pattern

• This hints at something different being allowed for classes
• We can match instance attributes!
• match event:
  ```python
case Click(x=x, y=y) if x < y:
    print("Lower-right click happened", x, y)
  ```
Class Pattern

• This syntax is a bit clunky and requires keyword-style attributes
• We can use `__match_args__` to specify the order of attributes instead:
  ```python
class Click:
    __match_args__ = ('x','y')

    match event:
      case Click(x,y) if x < y:
        print("Lower-right click happened", x, y)
  ```
• Dataclasses automatically order attributes based on their position
Matching Enumerated Values or Constants

- Can use dotted notation to reference the value of an enumerated value or constant (`Button.LEFT`)
- Cannot use bare identifiers (e.g. referencing constants) because they are interpreted as part of the pattern...
Mapping Pattern

- Just like matching sequences, we can also match mappings (i.e. dictionaries)
- Any unmatched key-value pairs are ignored
  - You don't need to use the multiple match as with sequences
  - But you can use **rest** if you want to use them
Mapping Pattern

• for action in actions:
  match action:
  case {"text": message, "color": c}:
    ui.set_text_color(c)
    ui.display(message)
  case {"sleep": duration}:
    ui.wait(duration)
  case {"sound": url, "format": "mp3"}:
    ui.play(url)
  case {"sound": _, "format": _}:
    warning("Unsupported audio format")
Match Statement Guidance

• Zen of Python: "There should be one-- and preferable only one --obvious way to do it."
  - Can use if/elif/else logic
  - Can use checks of sequence length, dictionary structure
• If you're emulating a switch statement, don't use a match statement
• If you're matching structure (sequence, mapping, object), a match statement may be a good idea
Example

```python
def eval_expr(expr):
    '''Evaluate an expression and return the result.'''
    match expr:
        case BinaryOp('+', left, right):
            return eval_expr(left) + eval_expr(right)
        case BinaryOp('-', left, right):
            return eval_expr(left) - eval_expr(right)
        case BinaryOp('*', left, right):
            return eval_expr(left) * eval_expr(right)
        case BinaryOp('/', left, right):
            return eval_expr(left) / eval_expr(right)
        case UnaryOp('+', arg):
            return eval_expr(arg)
        case UnaryOp('-', arg):
            return -eval_expr(arg)
        case VarExpr(name):
            raise ValueError(f"Unknown value of: {name}\")
        case float() | int():
            return expr
        case _:
            raise ValueError(f"Invalid expression value: {repr(expr)}")
```

[G. van Rossum]