Functions

6.1000 LECTURE 3

Previously (on the 6.1000 TV show)

- Elements of computation in Python
  - objects, operations, branching, looping
  - variables to keep track of obejcts
- Limitations
  - input/output through setting variables or terminal text
- Today
  - organize code into reusable sub-programs: functions
  - Python syntax for functions
  - semantics of calling functions
  - strategies for using functions, cool Python features

Status check

- Following class content
  - okay if not replicating class coding in real time
  - but be able to recreate the steps on your own
- Work on finger exercises early
- Ask for help if needed
  - office hours MTWR 11 am to 9 pm, F until 5 pm
  - instructor office hours Thu 1:30 pm or by appointment
- Last day to switch to 6.100A is next Tue 9/16

Last time: Divisibility by 3

- Mixing levels of conceptual detail
- Input limited to variable assignment or input()
  - have to manually edit line
  - error-prone
- Output limited to print() or other variable assignment
  - lack of reuse

Functions as contained programs

- Accept input through **parameters**
- Produce output through a `return` statement
- Body code is indented
  - hence need `pass` if empty

```
def is_oddly_even(num):
    div_by_2 = num % 2 == 0
    div_by_4 = num % 4 == 0
    return div_by_2 and not div_by_4
```

Using functions

- num is formal parameter
- def is_oddly_even(num): is function signature
- Call with a concrete argument is_oddly_even(10)
- Body code runs with num assigned to 10
- Body code stops when return False
- Function call at top level evaluates to return value

```python
def is_oddly_even(num):
    div_by_2 = num % 2 == 0
    div_by_4 = num % 4 == 0
    return div_by_2 and not div_by_4

result = is_oddly_even(10)
print(result)
```

Mechanism of function calls

1. Identify function object
2. Evaluate arguments in order
3. Set up frame/environment for function
4. Assign parameter names in frame
5. Run body with respect to that frame until return
   - *If reference any variables not in frame, look instead in the global frame*
6. Evaluate original function call as returned object

None

- None is a special value in Python
  - indicates absence of any meaningful value
  - needs to be represented by an actual object
- None is the only object of type `NoneType`
- All functions must return (i.e., evaluate to) some object
  - if body ends without encountering `return` statement, automatically returns None
- E.g., `print()` is called for **side-effect** of displaying text
  - doesn't affect objects in memory
  - evaluates to None
- **Beware:** None acts like `False` if used as a condition

Reorganizing code

Divisibility check

- Step 1: Put it in a function
- Call it with a single parameter for `upper_limit`
  - easier than hunting for the line to change
  - easy to call repeatedly on different inputs

Divisibility check v2

- Separate out helper functions
  - `add_digits(num)`
  - `check_rule_on_instance(num, digits_sum)`
- Now easier to improve each helper’s code on its own
  - better readability in main function
  - simplify Boolean logic in `check_rule_on_instance()`
- Consolidate `print()`s into main function

Divisibility check v3

- Separate checking code from reporting code
- Previous top-level function only returns `True` or `False`
  - no longer need `rule_holds` flag
  - early `return`s in functions simplify control flow

Divisibility check v4 [exercise]

- Extend v3 code to check divisibility rule for numbers beyond 3

```
for k in range(3, 10):
    report_divisibility_checks(k, 100)
```

Generalizing finding roots

- Public function
  - `find_root(num, ...)`
- Actual algorithm implementation
  - `bisection_root(...)`
- Sub-helper
  - `get_next_guess(lower, upper, ...)`

Generalizing finding roots

- **Docstrings**
  - explain functions’ purpose and specification
  - tells user how to use it, not how it works inside
  - accessible via `help()` on Python REPL
    - run `$ python3 -i file.py` to drop into REPL after running file

- **Conventions**
  - ***triple-quoted string*** as first statement in body
  - one-line summary of what it’s for
  - ***specify*** parameters’ types, purpose, restrictions
  - ***specify*** return type and meaning
  - https://peps.python.org/pep-0257/

Generalizing finding roots

- **Handle n-th roots**
  - additional power parameter
  - ultimately needed in get_next_guess()
  - needs to get passed through from find_root() call

Generalizing finding roots

- **Default parameter values**
  - don’t want to force user to always specify `power` and `epsilon`
  - default values can be specified in function signature
  - call `find_root(12345)`
    - `2` and `0.01` are automatically substituted in for power and epsilon
  - convention: no spaces around `=`

Generalizing finding roots

- **Keyword-specified arguments**
  - can specify arguments in function call by **keyword** that matches **formal parameter name**
  - `find_root(12345, epsilon=0.001)`
  - `power` still gets automatically assigned to **2**
  - keyword arguments can come in any order after positional arguments
  - also convention: no spaces around **=**

Generalizing finding roots

- **Multiple return values**
  - `get_next_guess()` returns both `guess` and evaluation of its power
    - in preparation for determining closeness
  - multiple values **separated by comma** in `return` statement
  - caller receives them by **comma-separated variables**
  - actual story is even cooler, wait until Lecture 7

Where we are

- Have all content needed to complete **Pset 1**
  - due next Wed 9/17
  - checkoffs start next Thu 9/18
- **Recitation** on Friday
  - group exercise in **reorganizing code** into functions
  - more practice with **bisection search**
- Can now abstract behaviors with functions. What about abstracting data?
  - Next couple weeks: **collections** of data