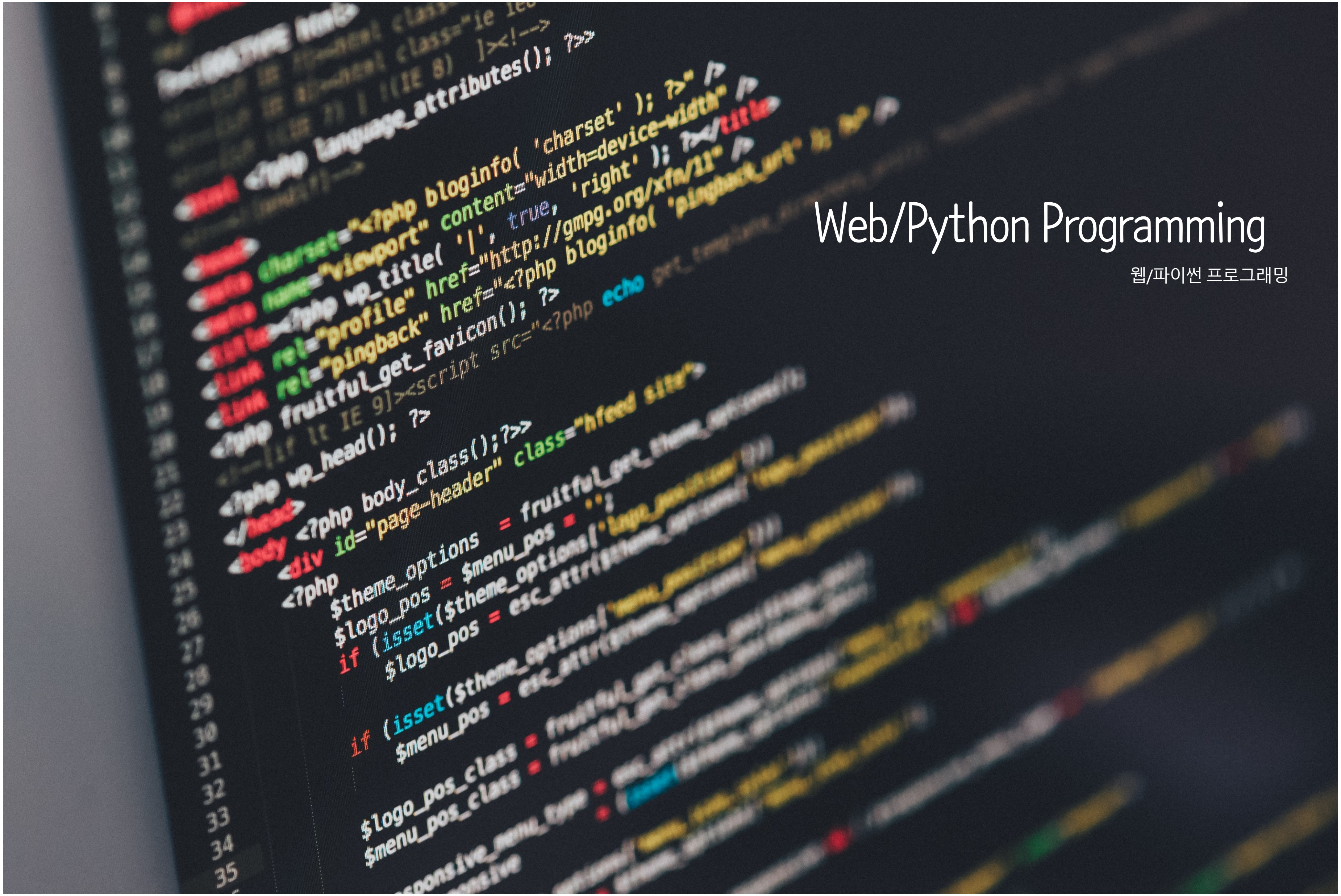


Web/Python Programming

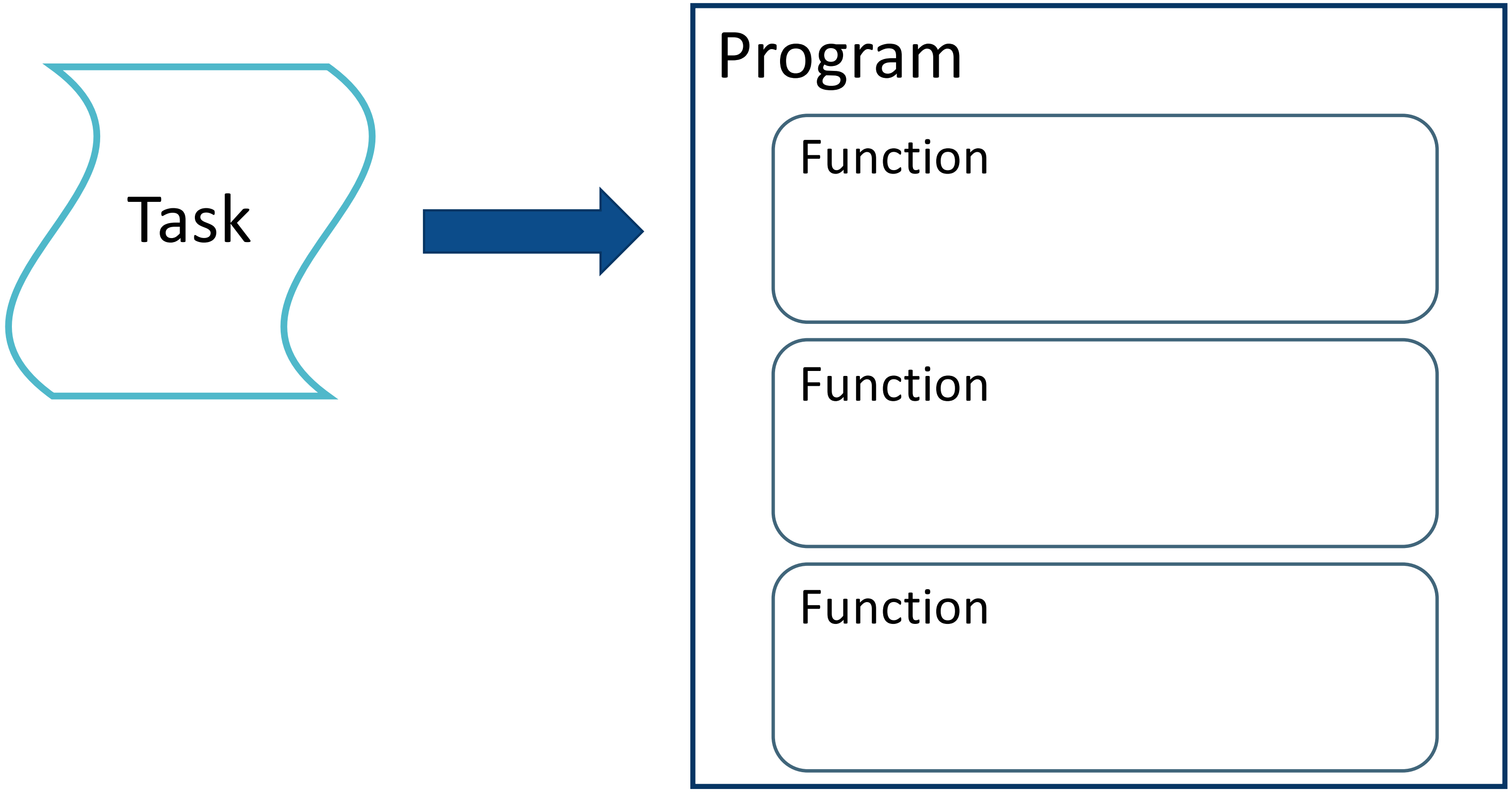
웹/파이썬 프로그래밍

Today

- Designing algorithms
  - Searching for the smallest values
  - Timing the functions

Algorithm

- An algorithm is a set of steps that accomplishes a task.



Top-down design: An algorithm writing technique

- Programmers often write algorithms in English + mathematics, and then translate it into Python.

- In mathematics, the first versions of “proofs” often handle common cases well, but fail for odd cases: mathematicians look for counterexamples.

- An implementation may look reasonable, but will contain bugs: Programmers test their code as they write it.

Searching for the smallest values

- How to find the index of the smallest items in an unsorted list
  - Find, remove, find
  - Sort, identify minimums, get indices
  - Walk through the list

- Suppose we have data showing number of humpback whales sighted off the coast of British Columbia over the past ten years:

```
>>> counts = [809,834,477,478,307,122,96,102,324,476]
>>> min(counts)
96
>>> low = min(counts)
>>> min_index = counts.index(low)
>>> print(min_index)
6
>>> counts.index(min(counts))
6
```

Find, remove, find

```
>>> counts = [809,834,477,478,307,122,96,102,324,476]
```

- Find the indices of the two smallest values?

- Find the index of the minimum, and save the first index.
- Remove that item from the list.
- Find the index of the new minimum item in the list, and save the second index.
- Put back the value we removed and adjust the second index to account for that reinsertion.

Sort, identify minimums, get indices

```
>>> counts = [809,834,477,478,307,122,96,102,324,476]
```

- Find the indices of the two smallest values?

- Sort the list, and get the two smallest numbers.
- Find their indices in the original list.

Walk through the list

```
>>> counts = [809,834,477,478,307,122,96,102,324,476]
```

- Find the indices of the two smallest values?

- Examine each value in the list in order
- Keep track of the two smallest values found so far
- Update these two values when a new smaller value is found

Find, remove, find

```python
def find_two_smallest(L):
    """ (list of float) -> tuple of (int, int)
    Return a tuple of the indices of the two smallest values in list L.
    >>> find_two_smallest([809, 834, 477, 478, 307, 122, 96, 102, 324, 476])
    (6, 7)
    """
    # Get the minimum item in L
    # Find the index of that minimum item
    # Remove that item from the list
    # Find the index of the new minimum item in the list
    # Put the smallest item back in the list
    # If necessary, adjust the second index
    # Return the two indices
```

Sort, identify minimums, get indices

```python
def find_two_smallest(L):
    """ (list of float) -> tuple of (int, int)
    Return a tuple of the indices of the two smallest values in list L.
    >>> find_two_smallest([809, 834, 477, 478, 307, 122, 96, 102, 324, 476])
    (6, 7)
    """
    # Sort a copy of L
    # Get the two smallest numbers
    # Find their indices in the original list L
    # Return the two indices
```

Walk through the list

```python
def find_two_smallest(L):
    """ (list of float) -> tuple of (int, int)
    Return a tuple of the indices of the two smallest values in list L.
    >>> find_two_smallest([809, 834, 477, 478, 307, 122, 96, 102, 324, 476])
    (6, 7)
    """
    # Examine each value in the list in order
    # Keep track of the indices of the two smallest values found so far
    # Update these values when a new smaller value is found
    # Return the two indices
```

Walk through the list

```
def find_two_smallest(L):
    """ (list of float) -> tuple of (int, int)
    Return a tuple of the indices of the two smallest values in list L.
    >>> find_two_smallest([809, 834, 477, 478, 307, 122, 96, 102, 324, 476])
    (6, 7)
    """
    # Set min1 and min2 to the indices of the smallest and next-smallest
    # values at the beginning of L
    # Examine each value in the list in order
    #     Update these values when a new smaller value is found
    # Return the two indices
```

Timing the functions

- Profiling a program

- Time: how long it takes to run
- Space: how much memory it uses

- Fast programs are more useful than slow ones
- Programs that need less memory are more useful than ones that need more memory

- One way to time how long code takes to run

Timing the functions: example

- Three `find_two_smallest` functions
- We will use those functions to find the two lowest values in a list on 1400 monthly readings of air pressure in Darwin, Australia, from 1882 to 1998.
- Module `time` contains `perf_counter` function, which returns a time in seconds. We call it before and after the code we want to time and take the difference to find out how many seconds have elapsed.

```
>>> import time
>>> t1 = time.perf_counter()
>>> # Some code runs here
>>> t2 = time.perf_counter()
```

```
>>> t1
87.2160792833727
>>> t2
97.1451065977857
>>> print("{:.2f} sec".format(t2-t1))
9.93 sec
```

Timing the functions: example

```
find_remove_find took 0.00 seconds.
sort_identify took 0.00 seconds.
walk_through took 0.00 seconds.
```

- Not much difference!!
- Simplicity or clarity > speed

```
find_remove_find took 0.27 ms.
sort_identify took 1.06 ms.
walk_through took 1.11 ms.
```

- But what if we want to process millions of values instead of 1400?

Summary

- The most effective way to design algorithms is to use top-down design, in which goals are broken down into subgoals until the steps are small enough to be translated directly into a programming language.
- Almost all problems have more than one correct solution. Choosing between them often involves a trade-off between simplicity and performance.
- The performance of a program can be characterized by how much time and memory it uses. This can be determined experimentally by profiling its execution. One way to profile time is with function `perf_counter` from module `time`.