1.4 Python Type Annotations

- Variable types never change!
- Only objects have a Type.

PYTHON TYPES:

PRIMITIVE TYPES:

- We use their type names (in type contract)
- None is referred to as value and type

Type name	Sample values
`int`	`0`, `148`, `-3`
`float`	`4.53`, `2.0`, `-3.49`
`str`	`'hello world'`, `''`
`bool`	`True`, `False`
`None`	`None`

COMPOUND TYPES:

- We use their type names (in type contract) & specify their primitive types

- What type do the list, dict, tuple take? (That's where we are more specific)
- Typing module : expresses the more detailed types (primitive)

Type	Description	Example
`List[T]`	a list whose elements are all of type `T`	`[1, 2, 3]` has type `List[int]`
`Dict[T1, T2]`	a dictionary whose keys are of type `T1` and whose values are of type `T2`	`{'a': 1, 'b': 2, 'c': 3}` has type `Dict[str, int]`
`Tuple[T1, T2, ...]`	a tuple whose first element has type `T1`, second element has type `T2`, etc.	`('hello', True, 3.4)` has type `Tuple[str, bool, float]`

We can nest these type expressions within each other; for example, the nested list `[[1, 2, 3], [-2]]` has type `List[List[int]]`.

ANNOTATING FUNCTIONS:

- `def can_divide(num: int, divisor: int) -> bool:`
  - Annotate the type of parameters after the semicolon
  - Annotate the return type after the arrow
  - To use compound types import typing module

ANNOTATING INSTANCE ATTRIBUTES:

- Attributes and their types go in the body of the class , at the top: after docstrings & before methods.

```
from typing import Dict, Tuple


class Inventory:
    """The inventory of a store.

    Keeps track of all of the items available for sale in the store.

    Attributes:
        size: the total number of items available for sale.
        items: a dictionary mapping an id number to a tuple with the
            item's description and number in stock.
    """
    size: int
    items: Dict[int, Tuple[str, int]]

    ...  # Methods omitted
```

ANNOTATING METHODS:

- Self is not annotated: its the class that this method belongs to
- When the class is the type of another parameter/ the return type of a method : Include this import statement at the top of the python file.
- Let's say you haven't defined a function yet, you are saying that you will define it eventually.

```python
# This is the special import we need for class type annotations.
from __future__ import annotations


class Inventory:

    # The type of self is omitted.
    def __init__(self) -> None:
        ...
    def add_item(self, item: str, quantity: int) -> None:
        ...
    def get_stock(self, item: str) -> int:
        ...

    def compare(self, other: Inventory) -> bool:
        ...
    def copy(self) -> Inventory:
        ...
    def merge(self, others: List[Inventory]) -> None:
        ...
```

4 ADVANCED TYPES:

****Imported from the python module****

ANY:

- The type of the function can be anything
- Don't overuse it... defeats purpose of annotations = be specific!

```python
from typing import Any


# This function could return a value of any type
def get_first(items: list) -> Any:
    return items[0]
```

UNION:

- The value can be one of two different types

```python
from typing import Union


def cube_root(x: Union[int, float]) -> float:
    return x ** (1/3)
```

OPTIONAL:

- Shorter way, Instead of using union (equivalent to `Union[Type, None]`)
- The value can be a certain type or none.

```python
from typing import Optional


def find_pos(numbers: List[int]) -> Optional[int]:
    """Return the first positive number in the given list.

    Return None if no numbers are positive.
    """
    for n in numbers:
        if n > 0:
            return n
```

CALLABLE:

- The type of a parameter, return value / instance value is a function
- Two expressions in square brackets: (Basically the type contract for that variable; which is a function)
  1. List of the arguments types
  2. Return type