Function Types and Higher-order Functions

Today starts as a Paper + Pencil or Tablet + Pencil day... please keep laptops stowed away!

COMP110 - CL15

2024/03/26

Warm-up: Fill in the blanks with valid function call names.

This will be today's attendance submission (CL15). Submit once complete.

```python
def e(x: int, y: int) -> int:
    return x + y


def f(x: int) -> int:
    return 2 * x


def g(x: str) -> int:
    return len(x)


def h(x: int) -> str:
    return str(x)
```

```python
a: str = ______(110)


b: int = ______(110)


c: int = ______(100, 10)


d: int = ______("110")
```

The *Type* of a Function

Given the following functions, write each's Callable type

```python
def e(x: int, y: int) -> int:
    return x + y


def f(x: int) -> int:
    return 2 * x


def g(x: str) -> int:
    return len(x)


def h(x: int) -> str:
    return str(x)
```

Example for e:

`Callable[[int, int], int]`

#2: Diagram the Listing (in diagram, ignore the import)

```
1   from typing import Callable
2
3
4   def f(x: int) -> str:
5       return str(2 * x)
6
7
8   g: Callable[[int], str]
9   g = f
10
11  result: str = g(110)
12  print(result)
```

#3: Fill in the blank with all possible, valid function names:

```
4   def e(x: int) -> int:
5       return 2 // x
6
7
8   def f(x: int) -> str:
9       return str(2 * x)
10
11
12  def g(x: int) -> int:
13      return 2 * x
14
15
16  def h(x: int) -> int:
17      return 2 + x
18
19
20  a_function: Callable[[int], int] = _____________
21
22  print(a_function(110))
```

Function Writing Practice

- Start a new file in your lecture directory, add directory named cl15, create a Python module named higher_order.py
- Write two functions:
  - A function named *square* that takes a float and returns a float, squaring it
  - A function named *map_square* that takes a list of floats, and returns a new list of floats where each item is the result of *calling* your *square* function with the item.
- Try calling each function in the Trailhead REPL to confirm correctness

Code Along

- Now let's try doing the same for *double* and *map_double* functions

- Can you identify some similarities?

- Besides the name of the function, what is the only difference?

- Let's try *abstracting* this algorithm to a single function called *map* with a *Callable* parameter.

```python
def map_square(xs: list[float]) -> list[float]:
    result: list[float] = []
    for x in xs:
        result.append(square(x))
    return result


def map_double(xs: list[float]) -> list[float]:
    result: list[float] = []
    for x in xs:
        result.append(double(x))
    return result
```

Goal

- Notice we've parameterized the function that gets called to a *Callable* parameter named ***f***

The Higher-order *map* Function Definition

```python
def map(f: Callable[[float], float], xs: list[float]) -> list[float]:
    result: list[float] = []
    for x in xs:
        result.append(f(x))
    return result
```

Generalizes these more narrowly useful functions.

```python
# def map_square(xs: list[float]) -> list[float]:
#     result: list[float] = []
#     for x in xs:
#         result.append(square(x))
#     return result


# def map_double(xs: list[float]) -> list[float]:
#     result: list[float] = []
#     for x in xs:
#         result.append(double(x))
#     return result
```

Now *any* function that takes a float and returns a float can be *mapped* over a list of floats.

```python
values: list[float] = [1.0, 5.0, 10.0]
print(map(double, values)) # [2.0, 10.0, 20.0]
print(map(square, values)) # [1.0, 25.0, 100.0]
```

Diagram

```python
from typing import Callable


def map(f: Callable[[float], float], xs: list[float]) -> list[float]:
    result: list[float] = []
    for x in xs:
        result.append(f(x))
    return result


def halve(x: float) -> float:
    return x / 2.0


values: list[float] = [2.0, 4.0]
ys: list[float] = map(halve, values)
print(ys)
```

```python
from typing import Callable


def map(f: Callable[[float], float], xs: list[float]) -> list[float]:
    result: list[float] = []
    for x in xs:
        result.append(f(x))
    return result


def halve(x: float) -> float:
    return x / 2.0


values: list[float] = [2.0, 4.0]
ys: list[float] = map(halve, values)
print(ys)
```