elif, Constants , Heap IDs, and more on Parameters

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

COMP110 - CL06

2024/02/06

Announcements

- EX02 Grade Calculator - Due Wednesday 2/14
- Office Hours Closed Saturday 2/10 through Tuesday 2/13 - Well-being
- QZ01 - Thursday 2/15

Warm-up: What is the printed output?

```python
def pack(degrees_fahrenheit: float) -> str:
    if degrees_fahrenheit <= 32.0:
        return "Warm Jacket"
    else:
        if degrees_fahrenheit == 0.0:
            return "REALLY Warm Jacket"
        else:
            if degrees_fahrenheit > 60.0:
                return "Long Sleeve Shirt"
            else:
                if degrees_fahrenheit > 75.0:
                    return "Short Sleeve Shirt"
                else:
                    return "Tank Top"


print(pack(degrees_fahrenheit=0.0))
print(pack(degrees_fahrenheit=95.0))
print(pack(degrees_fahrenheit=55.0))
```

Warning: Illogical conditional statements can lead to *unreachable code*

```python
def pack(degrees_fahrenheit: float) -> str:
    if degrees_fahrenheit <= 32.0:
        return "Warm Jacket"
    else:
        if degrees_fahrenheit == 0.0:
            return "REALLY Warm Jacket"
        else:
            if degrees_fahrenheit > 60.0:
                return "Long Sleeve Shirt"
            else:
                if degrees_fahrenheit > 75.0:
                    return "Short Sleeve Shirt"
                else:
                    return "Tank Top"
```

Notice the first if-then statement will be processed for any value less than or equal to 0.0, and in the else branch we test for the same value equal to 0.0.

As written, *no value* for `degrees_fahrenheit` will return "REALLY Warm Jacket" being returned. This is ***unreachable code.***

What other return statements are **unreachable?**

Rewrite the nested if-else statements to be *more logical* and *easier to reason about*.

```python
def pack(df: float) -> str:
    """Packing advice."""
    if df <= 50.0:
        return "Warm Jacket"
    else:
        if df <= 0.0:
            return "Stay Inside"
        else:
            if df >= 75.0:
                return "Short Sleeves"
            else:
                return "Long Sleeves"
```

Using the `elif` statement

The following two code snippets are *semantically* equivalent.

```python
def pack(degrees: float) -> str:
    """Packing advice."""
    if degrees <= 0.0:
        return "Stay Inside"
    else:
        if degrees <= 50.0:
            return "Warm Jacket"
        else:
            if degrees < 75.0:
                return "Long Sleeves"
            else:
                return "Short Sleeves"
```

```python
def pack(degrees: float) -> str:
    """Packing advice."""
    if degrees <= 0.0:
        return "Stay Inside"
    elif degrees <= 50.0:
        return "Warm Jacket"
    elif degrees < 75.0:
        return "Long Sleeves"
    else:
        return "Short Sleeves"
```

Warm-up Part 2: What is the printed output?

```python
def pack(degrees: float) -> str:
    """Packing advice."""
    if degrees <= 0.0:
        return "Stay Inside"
    elif degrees > 32.0:
        return "Warm Jacket"
    elif degrees >= 65.0:
        return "Long Sleeves"
    else:
        return "Short Sleeves"


print(pack(degrees=32.0))
print(pack(degrees=95.0))
```

Named Constants

Putting a Name to "Magical Values"

- Programs often involve *constant values* in computations and other places
  - For example: π, e, SALES_TAX, GAME_TITLE, FOOT_IN_INCHES and so on
- Rather than sprinkling *literal values* for these constants in *many places* through a program, often called "Magic Numbers", defining **named constants** is encouraged
- By convention, named constants are ALL_CAPITAL_LETTERS with multiple words separated by underscores.
- For example:
  - PI: float = 3.14159
  - SALES_TAX: float = 0.07
- When defined at the *global level* the named constant is available throughout your Python module. When defined inside a function, at a *local level,* the named constant is only defined in the function.
  - Why? … *Name resolution rules!*

Tuple Concatenation

Like string values, two tuples can be concatenated to form a new, larger tuple.

- (1,) + (2,) evaluates to (1, 2)
- () + (1, 2) evaluates to (1, 2)
- (1, 2) + (3,) evaluates to (1, 2, 3)
- (110,) + (101,) evaluates to (110, 101)

- The operand tuples remain *unchanged*, the resulting tuple is a *new object*.

Diagram the following program

```python
"""Demonstration of named constants."""

ZERO: float = 0.0
ORIGIN_1D: tuple[float] = (ZERO,)
ORIGIN_2D: tuple[float, float] = ORIGIN_1D + ORIGIN_1D


def distance(a: tuple[float, float], b: tuple[float, float]) -> float:
    """Calculate the distance between two points."""
    SQRT_EXP: float = 0.5
    return ((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2) ** SQRT_EXP


print(distance(a=ORIGIN_2D, b=(0.0, 2.0)))
```

What are the diagrammed arrows, anyway?

Memory Addresses!

- Every object in a running Python program has a numerical identifier (**id**)
  - The built-in **id()** function will tell you an object's id
  - These id's are actually *memory addresses* representing *where* in memory a particular value is found. The details of memory addresses are beyond our concerns, but it's worth noting!
- Notice these numbers are *very large... trillions!*
- Arrows are a simplification of *id*

```python
"""Demonstration of named constants."""

ZERO: float = 0.0
ORIGIN_1D: tuple[float] = (ZERO,)
ORIGIN_2D: tuple[float, float] = ORIGIN_1D + ORIGIN_1D


def distance(a: tuple[float, float], b: tuple[float, float]) -> float:
    """Calculate the distance between two points."""
    SQRT_EXP: float = 0.5
    return ((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2) ** SQRT_EXP


print(distance(a=ORIGIN_2D, b=(0.0, 2.0)))
```

id(ORIGIN_1D)

281472890902976 (int)

id(ORIGIN_2D)

281472891435904 (int)

id(distance)

281472890217120 (int)

Moving Forward: Diagrams with Heap IDs

- Moving forward, when objects are added to the heap (eg. functions, tuples, and more soon) number each item with a boxed Heap ID starting from **`0`** and counting up
- When referring to an object on the heap, rather than drawing an arrow, write: "`id:0`"
- When *accessing* or *reading* a name that holds a Heap ID, look up its value in the heap in order to know what to do with it

Diagram the following program with Heap IDs

No arrows needed! Just record the Heap ID in the stack value as <u>id:X</u> where X is object's Heap ID.

```python
UNIT_POINT: tuple[float, float] = (1.0, 1.0)


def add(a: tuple[float, float], b: tuple[float, float]) -> tuple[float, float]:
    """Add two 2D point tuples."""
    return (a[0] + b[0], a[1] + b[1])


print(add(a=UNIT_POINT, b=UNIT_POINT))
```

Parameters and Arguments

Keyword Arguments

Positional Arguments

Default Parameters

Trace the Following Program with Heap IDs

```python
def gen(stop: int, acc: tuple[int, ...] = (), i: int = 0) -> tuple[int, ...]:
    """Generate a tuple from i to stop."""
    if i >= stop:
        return acc
    else:
        return gen(stop, acc + (i,), i + 1)


print(gen(3))
```