1 Control

Control structures direct the flow of logic in a program. For example, conditionals (if-elif-else) allow a program to skip sections of code, while iteration (while), allows a program to repeat a section.

If statements

Conditional statements let programs execute different lines of code depending on certain conditions. Let’s review the if-elif-else syntax. Recall the following points:

- The else and elif clauses are optional, and you can have any number of elif clauses.
- A conditional expression is a expression that evaluates to either a true value (True, a non-zero integer, etc.) or a false value (False, 0, None, "", [], etc.).
- Only the suite that is indented under the first if/elif with a conditional expression evaluating to a true value will be executed.
- If none of the conditional expressions evaluate to a true value, then the else suite is executed. There can only be one else clause in a conditional statement!

Boolean Operators

Python also includes the boolean operators and, or, and not. These operators are used to combine and manipulate boolean values.

- not returns the opposite truth value of the following expression.
- and stops evaluating any more expressions (short-circuits) once it reaches the first false value and returns it. If all values evaluate to a true value, the last value is returned.
- or short-circuits at the first true value and returns it. If all values evaluate to a false value, the last value is returned.
Questions

1.1 Alfonso will only wear a jacket outside if it is below 60 degrees or it is raining.

Write a function that takes in the current temperature and a Boolean value telling if it is raining and returns True if Alfonso will wear a jacket and False otherwise.

This should only take one line of code!

def wears_jacket(temp, raining):
    """
    >>> wears_jacket(90, False)
    False
    >>> wears_jacket(40, False)
    True
    >>> wears_jacket(100, True)
    True
    """

1.2 To handle discussion section overflow, TAs may direct students to a more empty section that is happening at the same time.

Write a function that takes in the number of students in two sections and prints out what to do if either section exceeds 30 students.

def handle_overflow(s1, s2):
    """
    >>> handle_overflow(27, 15)
    No overflow
    >>> handle_overflow(35, 29)
    Move to Section 2: 1
    >>> handle_overflow(20, 32)
    Move to Section 1: 10
    >>> handle_overflow(35, 30)
    No space left in either section
    """
While loops

Iteration lets a program repeat statements multiple times. A common iterative block of code is the **while loop**.

As long as `<conditional clause>` evaluates to a true value, `<body of statements>` will continue to be executed. The conditional clause gets evaluated each time the body finishes executing.

Questions

1.3 What is the result of evaluating the following code?

```python
def square(x):
    return x * x
def so_slow(num):
    x = num
    while x > 0:
        x = x + 1
    return x / 0

square(so_slow(5))
```

1.4 Write a function that returns `True` if `n` is a prime number and `False` otherwise. After you have a working solution, think about potential ways to make your solution more efficient.

**Hint**: use the `%` operator: `x % y` returns the remainder of `x` when divided by `y`.

```python
def is_prime(n):
    """
    >>> is_prime(10)
    False
    >>> is_prime(7)
    True
    """
```
2 Higher Order Functions

A higher order function (HOF) is a function that manipulates other functions by taking in functions as arguments, returning a function, or both.

Functions as Arguments

One way a higher order function can manipulate other functions is by taking functions as input (an argument). Consider this higher order function called negate.

negate takes in a function f and a number x. It doesn’t care what exactly f does, as long as f is a function, takes in a number and returns a number. Its job is simple: call f on x and return the negation of that value.

Questions

2.1 Write a function that takes in a function cond and a number n and prints numbers from 1 to n where calling cond on that number returns True.

def keep_ints(cond, n):
    """Print out all integers 1..i..n where cond(i) is true"

    >>> def is_even(x):
    ...         # Even numbers have remainder 0 when divided by 2.
    ...         return x % 2 == 0
    ...     return x % 2 == 0
    >>> keep_ints(is_even, 5)
    2
    4
    """
Functions as Return Values

Often, we will need to write a function that returns another function. One way to do this is to define a function inside of a function:

```python
def outer(x):
    def inner(y):
        ...
        return inner
    return inner
```

The return value of `outer` is the function `inner`. This is a case of a function returning a function. In this example, `inner` is defined inside of `outer`. Although this is a common pattern, we can also define `inner` outside of `outer` and still use the same `return` statement. However, note that in this second example (unlike the first example), `inner` doesn’t have access to variables defined within the `outer` function, like `x`.

Questions

2.2 Use this definition of `outer` to fill in what Python would display when the following lines are evaluated.

```python
>>> def outer(n):
...     def inner(m):
...         return n - m
...     return inner

>>> outer(61)
```

```python
>>> f = outer(10)
>>> f(4)
```

```python
>>> outer(5)(4)
```
Write a function similar to `keep_ints` like before, but now it takes in a number \( n \) and returns a function that has one parameter \( \text{cond} \). The returned function prints out numbers from 1 to \( n \) where calling \( \text{cond} \) on that number returns True.

```python
def keep_ints(n):
    """Returns a function which takes one parameter \( \text{cond} \) and prints out all integers 1..i..n where calling \( \text{cond}(i) \) returns True."""

>>> def is_even(x):
...     # Even numbers have remainder 0 when divided by 2.
...     return x % 2 == 0

>>> keep_ints(5)(is_even)
2
4
"""
3 Environment Diagrams

An environment diagram keeps track of all the variables that have been defined and the values they are bound to.

\[ x = 3 \]

```python
def square(x):
    return x ** 2
```

square(2)

When you execute assignment statements in an environment diagram (like \( x = 3 \)), you need to record the variable name and the value:

1. Evaluate the expression on the right side of the = sign
2. Write the variable name and the expression’s value in the current frame.

When you execute def statements, you need to record the function name and bind the function object to the name.

1. Write the function name (e.g., `square`) in the frame and point it to a function object (e.g., `func square(x) [parent=Global]`). The [parent=Global] denotes the frame in which the function was defined.

When you execute a call expression (like `square(2)`), you need to create a new frame to keep track of local variables.

1. Draw a new frame. \(^a\) Label it with
   - a unique index (f1, f2, f3 and so on)
   - the intrinsic name of the function (`square`), which is the name of the function object itself. For example, if the function object is `func square(x) [parent=Global]`, the intrinsic name is `square`.
   - the parent frame ([parent=Global])
2. Bind the formal parameters to the arguments passed in (e.g. bind x to 3).
3. Evaluate the body of the function.

If a function does not have a return value, it implicitly returns None. Thus, the “Return value” box should contain None.

\(^a\)Since we do not know how built-in functions like `add(...)` or `min(...)` are implemented, we do not draw a new frame when we call built-in functions.
Questions

3.1 Draw the environment diagram that results from executing the code below.

```python
da = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

3.2 Draw the environment diagram that results from executing the code below. What will be the output (note this separately from the diagram)?

```python
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```