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.

1.1 If statements

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

```python
if <conditional expression>:
    <suite of statements>
elif <conditional expression>:
    <suite of statements>
else:
    <suite of statements>
```

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 are True, then the else suite is executed.
There can only be one else clause in a conditional statement!

1.2 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 True, the last value is returned.
- **or** short-circuits at the first True value and returns it. If all values evaluate to False, the last value is returned.

```python
>>> not None
True
>>> not True
False
>>> -1 and 0 and 1
0
>>> False or 9999 or 1/0
9999
```

1.3 Questions

1. Alfonso will only wear a jacket outside if it is below 60 degrees or it is raining. Fill in the function `wears_jacket` which 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!
   ```python
def wears_jacket(temp, raining):
    """
    >>> rain = False
    >>> wears_jacket(90, rain)
    False
    >>> wears_jacket(40, rain)
    True
    >>> wears_jacket(100, True)
    True
    """
```
2. To handle discussion section overflow, TA’s may direct students to a more empty section that is happening at the same time. Write the function `handle_overflow`, which takes in the number of students at two sections and prints out what to do if either section exceeds 30 students. See the doctests below for the behavior.

```python
def handle_overflow(s1, s2):
    """
    >>> handle_overflow(27, 15)
    No overflow.
    >>> handle_overflow(35, 29)
    1 spot left in Section 2.
    >>> handle_overflow(20, 32)
    10 spots left in Section 1.
    >>> handle_overflow(35, 30)
    No space left in either section.
    """
```

### 1.4 While loops

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

```
while <conditional clause>:
    <body of statements>
```

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

1. 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))
```

2. Fill in the `is_prime` function, which returns `True` if \( n \) is a prime number and `False` otherwise.
   
   **Hint:** use the `%` operator: \( x \ % \ y \) returns the remainder of \( x \) when divided by \( y \).
   
   ```python
def is_prime(n):
```
1. Implement `fizzbuzz(n)`, which prints numbers from 1 to `n` (inclusive). However, for numbers divisible by 3, print “fizz”. For numbers divisible by 5, print “buzz”. For numbers divisible by both 3 and 5, print “fizzbuzz”.

This is a standard software engineering interview question, but even though we’re barely one week into the course, we’re confident in your ability to solve it!

```python
def fizzbuzz(n):
    """
    >>> result = fizzbuzz(16)
    1
    2
    fizz
    4
    buzz
    fizz
    7
    8
    fizz
    buzz
    11
    fizz
    13
    14
    fizzbuzz
    16
    >>> result is None
    True
    """
```
2. Fill in the `choose` function, which returns the number of ways to choose \( k \) items from \( n \) items. Mathematically, \( \text{choose}(n, k) \) is defined as:

\[
\frac{n \times (n - 1) \times (n - 2) \times \cdots \times (n - k + 1)}{k \times (k - 1) \times (k - 2) \times \cdots \times 2 \times 1}
\]

```python
def choose(n, k):
    """Returns the number of ways to choose K items from N items."
    >>> choose(5, 2)
    10
    >>> choose(20, 6)
    38760
    """
```

## 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.

### 2.1 Functions as Arguments

One way a higher order function can exploit other functions is by taking functions as input. Consider this higher order function called `negate`.

```python
def negate(f, x):
    return -f(x)
```

`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.
2.2 Questions

1. Here are some possible functions that can be passed through as f.
   ```python
def square(n):
    return n * n

def double(n):
    return 2 * n
```

What will the following Python statements output?
```python
>>> negate(square, 5)
>>> negate(double, -19)
>>> negate(double, negate(square, -4))
```

2. Implement a function `keep_ints`, which takes in a function `cond` and a number `n`, and only prints a number from 1 to `n` if calling `cond` on that number returns `True`:
   ```python
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
    """
```
2.3 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
```

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 `inner` does not have access to variables defined within the outer function (whereas it does in the first example).

```python
def inner(y):
    ...

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

2.4 Questions

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

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

>>> outer(61)

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

>>> outer(5)(4)
```
2. Implement a function `keep_ints` like before, but now it takes in a number `n` and returns a function that has one parameter `cond`. The returned function prints out all numbers from 1..i..n where calling `cond(i)` returns True.

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