Control structures direct the flow of logic in a program. This can mean skipping a portion of code (conditionals) or repeating a portion of code multiple times (iteration).

1.1 Conditional Statements

Conditional statements let programs execute different lines of code depending certain conditions. The conditional statement in Python is an if-elif-else block:

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

Some notes:

- The else and elif statements are optional.
- You can have any number of elif statements.
- A conditional expression is a Python expression. All that matters for control is whether its value is a true value or a false value.
- The code that is executed is the suite that is indented under the first if/elif that has a true conditional expression. If none are true, then the else suite is executed.
- Once one suite is executed, the rest are skipped.
Note: in Python, there are a few things that are treated as false values:

- The boolean False
- The integer 0
- The value None
- And more... (we will learn about these later in the semester)

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

- `not True` evaluates to False, and `not False` evaluates to True.
- `True and True` evaluates to True, but a false value on either side makes it False.
- `False or False` evaluates to False, but a true value on either side makes it True.

1.2 Question

1. It’s lecture time! However, whether you go depends on certain conditions about timing, seats, and laziness. Write a simple function `which_lecture` that takes in inputs `time`, `seats_left`, `is_lazy` and prints out your decision.

   - `which_lecture` should print "go to lecture" if time is before 2:00pm, there are seats, and you are not lazy.
   - `which_lecture` should print "go to alt lecture" if time is after 2:00pm or there are no seats, and you are not lazy.
   - `which_lecture` should print "watch videos" if you feel lazy.

   `time` is in military format; e.g 2:20pm is 1420. `seats_left` is a non-negative integer. `is_lazy` is a boolean variable.

   ```python
def which_lecture(time, seats_left, is_lazy):
   ```
1.3 Iteration

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

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

This block of code states: “while the conditional clause is still True, continue executing the indented body of statements.” Here is an example:

```python
def countdown(x):
    while x > 0:
        print(x)
        x = x - 1
    print("Blastoff!")

def countdown(3)
```

1.4 Questions

1. 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` is divided by `y`.

   ```python
def is_prime(n):
```
2. Fill in the `choose` function, which returns the number of ways to choose \( k \) items from \( n \) items. Mathematically, `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 function that manipulates other functions is called a higher order function (HOF). A HOF can be a function that takes functions as arguments, returns a function, or both.

2.1 Functions as Argument Values

Suppose we want to square or double every natural number from 1 to \( n \) and print the result as we go. Fill in the functions \texttt{square\_every\_number} and \texttt{double\_every\_number} by using the \texttt{square} and \texttt{double} functions we have defined.

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

def square\_every\_number(n):
    
        """Prints out the square of every integer from 1 to \( n \).
        >>> square\_every\_number(3)
        1
        4
        9
        ""

    


def double(x):
    return 2 * x

def double\_every\_number(n):
    
        """Prints out the double of every integer from 1 to \( n \).
        >>> double\_every\_number(3)
        2
        4
        6
        ""

    

The only difference between \texttt{square\_every\_number} and \texttt{double\_every\_number} is the function called before printing (either \texttt{square} or \texttt{double}). Everything else is the same!
It would be nice to have a generalized function (let’s call it the `every` function) that took care of the `while` loop and the incrementing for us. That way, we could `triple_every_number` or `cube_every_number` without repeating so much code:

```python
def square_every_number(n):
    every(square, n)

def double_every_number(n):
    every(double, n)

def cube(x):
    return x * x * x

def cube_every_number(n):
    every(cube, n)
```

### 2.2 Questions

1. Implement the function `every` that takes in a function `func` and a number `n`, and prints the result of applying that function to each of the first `n` natural numbers.

   ```python
def every(func, n):
    """Prints out all integers from 1 to n with func applied on them."

    >>> def square(x):
    ... return x * x
    ... return x * x
    >>> every(square, 3)
    1
    4
    9
    """
```

2. Similarly, implement a function `keep`, which takes in a function `cond` and a number `n`, and only prints a number from 1 to `n` to the screen if calling `cond` on that number returns `True`:

   ```python
def keep(cond, n):
```
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
```

Note two things:

1. The return value of the outer function is inner! This is where a function returns a function.
2. In this case, the inner function is defined inside of the outer function. This is a common pattern, but it is not necessary — we could have defined inner outside of the outer and still keep the return statement the same.

2.4 Moar Questions

1. Write a function `and_add` that takes a function `f` (such that `f` is a function of one argument) and a number `n` as arguments. It should return a function that takes one argument, and does the same thing as the function `f`, except also adds `n` to the result.
**Discussion 2: Control and Higher Order Functions**

```python
def and_add(f, n):
    """Returns a new function. This new function takes an argument
    x and returns f(x) + n.
    """

    def new_f(x):
        return f(x) + n
    return new_f

>>> def square(x):
...     return x * x
... >>> new_square = and_add(square, 3)
... >>> new_square(4)  # 4 * 4 + 3
19
"""

2. The following code has been loaded into the python interpreter:

```python
def skipped(f):
    def g():
        return f
    return g
def composed(f, g):
    def h(x):
        return f(g(x))
    return h
def added(f, g):
    def h(x):
        return f(x) + g(x)
    return h
def square(x):
    return x*x
def two(x):
    return 2
```

What will python output when the following lines are evaluated?

```python
>>> composed(square, two)(7)
```

```python
>>> skipped(added(square, two))()(3)
```
3. Draw the environment diagram that results from running the following code.

```python
n = 7
def f(x):
    n = 8
    return x + 1
def g(x):
    n = 9
    return x + 3
def f(f, x):
    return f(f(x+2))
m = f(g, n)
```
4. Draw the environment diagram for the following code:

```python
from operator import add

def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f

make_adder = curry2(add)
add_three = make_adder(3)
five = add_three(2)
```