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** evaluate to a true value, 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 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.

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

### 1.3 Question

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):
    """
    >>> wears_jacket(90, False)
    False
    >>> wears_jacket(40, False)
    True
    >>> wears_jacket(100, True)
    True
    """
```
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 a true value, `<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. 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):
```
An environment diagram keeps track of all the variables that have been defined and the values they are bound to.

\[
x = 3
\]

\[
def \text{square}(x):
    \text{return } x ** 2
\]

\[
square(2)
\]

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 \(\text{square}(2)\)), you need to create a new frame to keep track of local variables.

1. Draw a new frame. 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.
2.1 Environment Diagram Question

1. Draw the environment diagram so we can visualize exactly how Python evaluates the code. What is the output of running this code in the interpreter?

```python
>>> from operator import add
>>> def sub(a, b):
...     sub = add
...     return a - b
>>> add = sub
>>> sub = min
>>> print(add(2, sub(2, 3)))
```
A **higher order function** (HOF) is a function that manipulates other functions by taking in functions as arguments, returning a function, or both.

### 3.1 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`.

```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.

### 3.2 Question

1. 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
...    >>> keep_ints(is_even, 5)
   2
   4
   " " "
```
3.3 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 (unlike the first example), `inner` doesn’t have access to variables defined within the `outer` function, like `x`.

```python
def outer(x):
    ...

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

3.4 Questions

1. 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)

>>> 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
    """
```

3. Write a function `and_add` that takes a one-argument function `f` 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.

```python
def and_add(f, n):
    """Return a new function. This new function takes an
    argument x and returns f(x) + n.
    >>> def square(x):
    ...     return x * x
    >>> new_square = and_add(square, 3)
    >>> new_square(4)    # 4 * 4 + 3
    19
    """
```