

# CONTROL AND ENVIRONMENT DIAGRAMS 1

---

COMPUTER SCIENCE 61A

June 23, 2016

---

## 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.

### 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:

```
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 an 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.

```
>>> 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!

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

**Solution:**

```
return temp < 60 or raining
```

2. To handle discussion section overflow, TAs 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.

```
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.  
    """
```

**Solution:**

```
if s1 <= 30 and s2 <= 30:  
    print("No overflow.")  
elif s2 > 30 and s1 < 30:  
    print(str(30 - s1) + "spots left in Section 1.")  
elif s1 > 30 and s2 < 30:  
    print(str(30 - s2) + "spots left in Section 2.")  
else:  
    print("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 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?

```
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))
```

**Solution:** Infinite loop because `x` will always be greater than 0; the `num / 0` is never executed.

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`.

```
def is_prime(n):
```

**Solution:**

```
    if n == 1:  
        return False  
    k = 2  
    while k < n:  
        if n % k == 0:  
            return False  
        k += 1  
    return True
```

Alternatively, the while loop's conditional expression could ensure that `k` is less than the square root of `n`.

## 1.6 Have Some More Control!

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!

```
def fizzbuzz(n):  
    """  
    >>> result = fizzbuzz(16)  
    1  
    2  
    fizz  
    4  
    buzz  
    fizz  
    7  
    8  
    fizz  
    buzz  
    11  
    fizz  
    13  
    14  
    fizzbuzz  
    16  
    >>> result == None  
    True  
    """
```

### Solution:

```
i = 1  
while i <= n:  
    if i % 3 == 0 and i % 5 == 0:  
        print('fizzbuzz')  
    elif i % 3 == 0:  
        print('fizz')  
    elif i % 5 == 0:  
        print('buzz')  
    else:
```

```
        print (i)
    i += 1
```

---

## 2 Lists and For Statements

---

### 2.1 List slicing and indexing

---

If we want to access more than one element of a list at a time, we can use a *slice*. Slicing a sequence is very similar to indexing. We specify a starting index and an ending index, separated by a colon. Python creates a new list with the elements from the starting index up to (but not including) the ending index. Specifically, we can write `[start:stop]` to slice a list with two integers.

*start* denotes the index for the beginning of the slice(inclusive)

*stop* denotes the index for the end of the slice(exclusive)

Using negative indices for start and end behaves in the same way as indexing into negative indices. Slicing a list always creates a new list.

```
>>> pizza = [1, 2, 3, 4]
>>> pizza[0]
1
>>> pizza[-1]
4
>>> pizza[-4]
1
>>> pizza[1:2]
[2]
>>> pizza[1:]
[2, 3, 4]
>>> pizza[-2:3]
[3]
```

### 2.2 For Statement Execution Procedure

---

```
for <name> in <expression>:
    <suite>
```

- Evaluate the header `<expression>`, which must yield an iterable value, such as a list
- For each element in that sequence, in order:
  - A. Bind `<name>` to that element in the current frame
  - B. Execute the `<suite>`

## 2.3 Questions

---

1. What would Python print?

```
>>> a = [1, 5, 4, [2, 3], 3]
>>> print(a[0], a[-1])
```

**Solution:**

1 3

```
>>> len(a)
```

**Solution:** 5

```
>>> 2 in a
```

**Solution:** False

```
>>> 4 in a
```

**Solution:** True

```
>>> a[3][0]
```

**Solution:** 2

2. What would Python print?

```
>>> apple = [3, 2, 1, 0]
>>> def banana(fruits):
    for apple in fruits:
        print(apple)
>>> banana(apple)
```

**Solution:**

3  
2  
1  
0

3. What would Python print?

```
>>> x = [1, 3, 5, 7]
>>> def partial(lst):
```



```
first = lst[0]
if first == 3:
    print('Hello')
else:
    print('Goodbye')
return lst
```

```
>>> partial(x)
```

**Solution:**

```
Goodbye
[1, 3, 5, 7]
```

## 4. What would Python print?

```
>>> lst = [3, 2, 1, 0]
>>> def fungus(spore):
    x = 0
    while spore[x] != 0:
        print('Mushroom!')
        x += 1
    return x
```

```
>>> fungus(lst)
```

**Solution:**

```
Mushroom!
Mushroom!
Mushroom!
3
```

5. Define a function `print_negative` that takes in a list `lst` and prints all the negative numbers in the list.

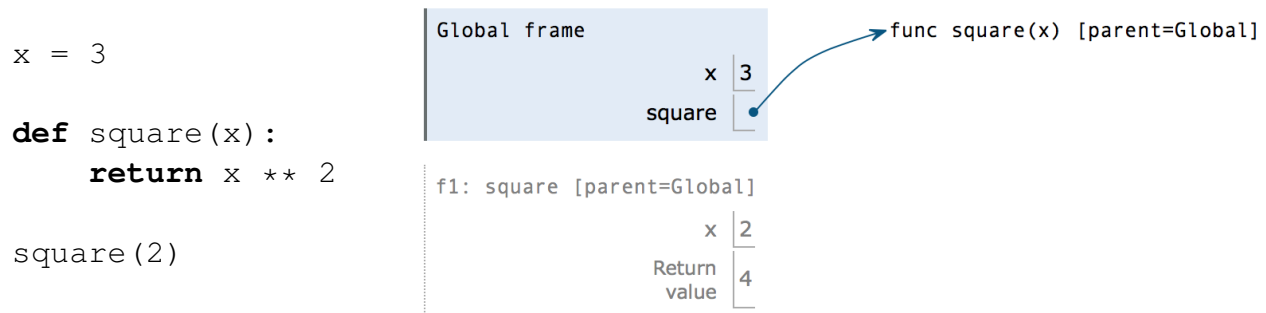
```
def print_negative(lst):
    for _____ in _____:
        if _____:
            print(_____)
```

**Solution:**

```
def print_negative(lst):  
    for x in lst:  
        if x < 0:  
            print(x)
```

### 3 Environment Diagrams

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



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. <sup>a</sup> 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`.

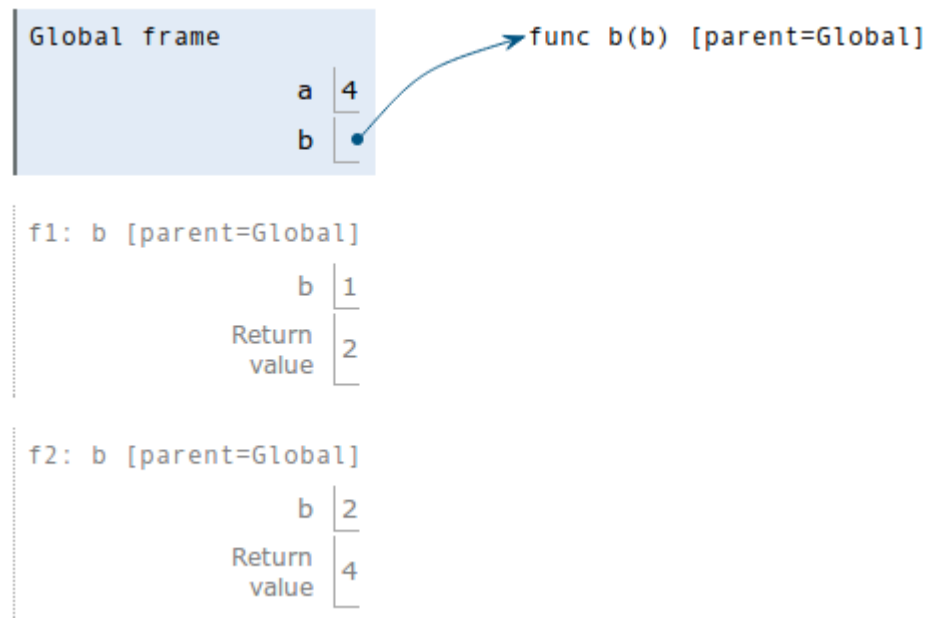
<sup>a</sup>Since we do not know how built-in functions like `add(...)` or `min(...)` are implemented, we do *not* draw a new frame when we call built-in functions.

### 3.1 Environment Diagram Questions

1. Draw the environment diagram that results from running the following code.

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

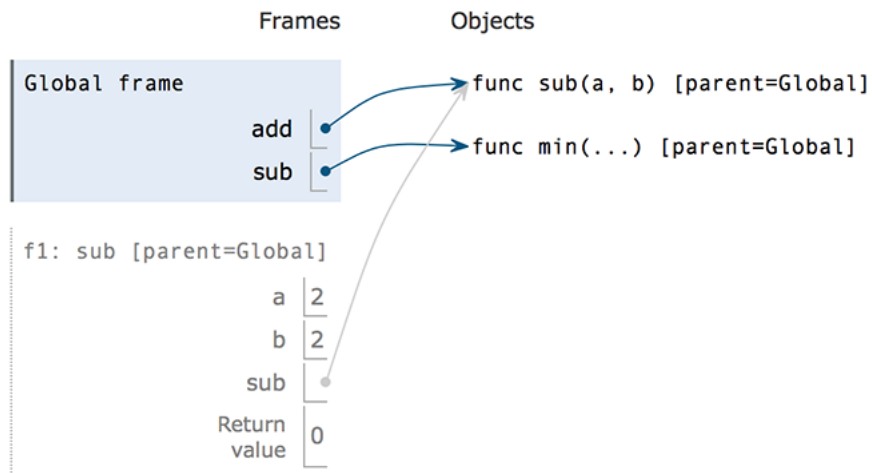
**Solution:**



2. 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?

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

**Solution:**



**Output:**

0