CS 61A Fall 2021

# Control, Environment Diagrams

Discussion 1: September 1, 2021 Solutions

## Control structures

**Control structures** direct the flow of a program using logical statements. For example, conditionals (`if-elif-else`) allow a program to skip sections of code, and iteration (`while`), allows a program to repeat a section.

## Conditional statements

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

- The `elif` and `else` clauses are optional, and you can have any number of `elif` clauses.
- A **conditional expression** is an expression that evaluates to either a truthy value (`True`, a non-zero integer, etc.) or a falsy value (`False`, `0`, `None`, `""`, `[]`, etc.).
- Only the **suite** that is indented under the first `if`/`elif` whose conditional expression evaluates 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.

Here's the general form:

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

## 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 boolean value of the following expression, and will always return either `True` or `False`.
- `and` evaluates expressions in order and stops evaluating (short-circuits) once it reaches the first falsy value, and then returns it. If all values evaluate to a truthy value, the last value is returned.
- `or` evalutes expressions in order and short-circuits at the first truthy value and returns it. If all values evaluate to a falsy value, the last value is returned.

For example:

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

### Q1: Jacket Weather?

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. This function should return `True` if Alfonso will wear a jacket and `False` otherwise.

First, try solving this problem using an `if` statement.

```
def wears_jacket_with_if(temp, raining):
    """
    >>> wears_jacket_with_if(90, False)
    False
    >>> wears_jacket_with_if(40, False)
    True
    >>> wears_jacket_with_if(100, True)
    True
    """
    if temp < 60 or raining:
        return True
    else:
        return False
```

Note that we'll either return `True` or `False` based on a single condition, whose truthiness value will also be either `True` or `False`. Knowing this, try to write this function using a single line.

```
def wears_jacket(temp, raining):
    return temp < 60 or raining
```

**Q2: Case Conundrum**

In this question, we will explore the difference between `if` and `elif`.

What is the result of evaluating the following code?

```
def special_case():
    x = 10
    if x > 0:
        x += 2
    elif x < 13:
        x += 3
    elif x % 2 == 1:
        x += 4
    return x

special_case()
```

What is the result of evaluating this piece of code?

```
def just_in_case():
    x = 10
    if x > 0:
        x += 2
    if x < 13:
        x += 3
    if x % 2 == 1:
        x += 4
    return x

just_in_case()
```

How about this piece of code?

```
def case_in_point():
    x = 10
    if x > 0:
        return x + 2
    if x < 13:
        return x + 3
    if x % 2 == 1:
        return x + 4
    return x

case_in_point()
```

Which of these code snippets result in the same output, and why? Based on your

findings, when do you think using a series of `if` statements has the same effect as using both `if` and `elif` cases?

The calls to `special_case` and `case_in_point` both return 12, while the call to `just_in_case` returns 19. Since the number 10 satisfies all three conditions in each function, the value of the variable `x` is incremented three times when `just_in_case` is called. A series of `if` statements has the same effect as using both `if` and `elif` cases if each `if` clause ends in a `return` statement.

### Q3: If Function vs Statement

Now that we've learned about how `if` statements work, let's see if we can write a function that behaves the same as an `if` statement.

```
def if_function(condition, true_result, false_result):
    """Return true_result if condition is a true value, and
    false_result otherwise.

    >>> if_function(True, 2, 3)
    2
    >>> if_function(False, 2, 3)
    3
    >>> if_function(3==2, 'equal', 'not equal')
    'not equal'
    >>> if_function(3>2, 'bigger', 'smaller')
    'bigger'
    """
    if condition:
        return true_result
    else:
        return false_result
```

Despite the doctests above, this function actually does *not* do the same thing as an `if` statement in all cases.

To demonstrate this, we want to find a case where this function will behave differently from an `if` statement. To do so in this problem, implement the following,

- `cond`: This should act as the "condition" of the "if".
- `true_func`: This should represent what we would want the result of the "if" to be in the case that the "condition" is truthy.
- `false_func`: This should represent what we would want the result of the "if" to be in the case that the "condition" is falsy.

so that `with_if_function` does *not* behave the same as `with_if_statement`, namely as specified in their doctests:

- When `with_if_statement` is called, we print out `61A`.
- When `with_if_function` is called, we print out both `Welcome to` and `61A` on separate lines.

Implement `cond`, `true_func`, and `false_func` below.

> **Hint**: If you are having a hard time identifying how `with_if_statement` and `with_if_function` would differ in behavior, consider the rules of evaluation for `if` statements and call expressions.

```
def with_if_statement():
    """
    >>> result = with_if_statement()
    61A
    >>> print(result)
    None
    """
    if cond():
        return true_func()
    else:
        return false_func()

def with_if_function():
    """
    >>> result = with_if_function()
    Welcome to
    61A
    >>> print(result)
    None
    """
    return if_function(cond(), true_func(), false_func())

def cond():
    return False

def true_func():
    print("Welcome to")

def false_func():
    print("61A")
```

The function `with_if_function` uses a call expression, which guarantees that all of its operand subexpressions will be evaluated before `if_function` is applied to the resulting arguments.

Therefore, even if `cond` returns `False`, the function `true_func` will be called. When we call `true_func`, we print out `Welcome to`. Then, when we call `false_func`, we will also print `61A`.

By contrast, `with_if_statement` will never call `true_func` if `cond` returns `False`. Thus, we will only call `false_func`, printing `61A`.

# While loops

To repeat the same statements multiple times in a program, we can use iteration. In Python, one way we can do this is with a **while loop**.

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

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

### Q4: Square So Slow

What is the result of evaluating the following code?

```
def square(x):
    print("here!")
    return x * x

def so_slow(num):
    x = num
    while x > 0:
        x=x+1
    return x / 0

square(so_slow(5))
```

**Hint:** What happens to `x` over time?

**Solution:** This program results in an infinite loop because `x` will always be greater than 0; `x / 0` is never executed. We also know that `here!` is never printed since the operand `so_slow(5)` must be evaluated before `function square(x)` can be called.

Here's a video walkthrough.

### Q5: Is Prime?

Write a function that returns `True` if a positive integer `n` is a prime number and `False` otherwise.

A prime number n is a number that is not divisible by any numbers other than 1 and n itself. For example, 13 is prime, since it is only divisible by 1 and 13, but 14 is not, since it is divisible by 1, 2, 7, and 14.

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

```
def is_prime(n):
    """
    >>> is_prime(10)
    False
    >>> is_prime(7)
    True
    """
    if n == 1:
        return False
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
```

**Q6: Fizzbuzz**

Implement the fizzbuzz sequence, which prints out a *single statement* for each number from 1 to `n`. For a number `i`,

- If `i` is divisible by 3 only, then we print "fizz".
- If `i` is divisible by 5 only, then we print "buzz".
- If `i` is divisible by both 3 and 5, then we print "fizzbuzz".
- Otherwise, we print the number `i` by itself.

Implement `fizzbuzz(n)` here:

```
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
    """
    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
```

To print something for each number from 1 to `n`, we can use a loop that goes through each number, and then check which of the cases applies using `if-elif-`

`else` to figure out what to print.

Students should be careful about the order in which they have their `if-elif` statements: we want to first check if `i` is divisible by both 3 and 5, or otherwise we will end up printing "fizz" if the student checked for divisibility by 3 first (or "buzz" if the student checked for divisibility by 5 first) rather than "fizzbuzz".

Video walkthrough

# Environment Diagrams

An **environment diagram** is a model we use to keep track of all the variables that have been defined and the values they are bound to. We will be using this tool throughout the course to understand complex programs involving several different assignments and function calls.

Here's a short program and its corresponding diagram:

See the web version of this resource for the environment diagram.

Remember that programs are mainly just a set of statements or instructions—so drawing diagrams that represent these programs also involves following sets of instructions! Let's dive in…

## Assignment Statements

Assignment statements, such as `x = 3`, define variables in programs. To execute one in an environment diagram, 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.

### Q7: Assignment Diagram

Use these rules to draw an environment diagram for the assignment statements below:

See the web version of this resource for the environment diagram.

We first assign `x` to the result of evaluating `11 % 4`. We then bind `y` to the current value of `x` (which we can figure out by looking it up in our current environment diagram). Finally, we'd like to update `x` to the new value that is the result of the current `x` squared.

Video walkthrough

## def Statements

A `def` statement creates ("defines") a function object and binds it to a name. To diagram `def` statements, record the function name and bind the function object to the name. It's also important to write the **parent frame** of the function, which is where the function is defined.

**A very important note:** Assignments for `def` statements use pointers to functions, which can have different behavior than primitive assignments (such as variables bound to numbers).

1. Draw the function object to the right-hand-side of the frames, denoting the intrinsic name of the function, its parameters, and the parent frame (e.g. `func square(x) [parent = Global]`.
2. Write the function name in the current frame and draw an arrow from the name to the function object.

### Q8: def Diagram

Use these rules for defining functions and the rules for assignment statements to draw a diagram for the code below.

See the web version of this resource for the environment diagram.

We first define the two functions `double` and `triple`, each bound to their corresponding name. In the next line, we assign the name `hat` to the function object that `double` refers to. Finally, we assign the name `double` to the function object that `triple` refers to.

Video walkthrough

# Call Expressions

**Call expressions**, such as `square(2)`, apply functions to arguments. When executing call expressions, we create a new frame in our diagram to keep track of local variables:

1. Evaluate the operator, which should evaluate to a function.
2. Evaluate the operands from left to right.
3. Draw a new frame, labelling it with the following:
   - A unique index (`f1`, `f2`, `f3`, …).
   - The **intrinsic name** of the function, 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`]).
4. Bind the formal parameters to the argument values obtained in step 2 ( e.g. bind `x` to 3).
5. Evaluate the body of the function in this new frame until a return value is obtained. Write down the return value in the frame.

If a function does not have a return value, it implicitly returns `None`. In that case, the "Return value" box should contain `None`.

**Note:** Since we do not know how built-in functions like `min(...)` or imported functions like `add(...)` are implemented, we do not draw a new frame when we call them, since we would not be able to fill it out accurately.

### Q9: Call Diagram

Let's put it all together! Draw an environment diagram for the following code. You may not have to use all of the blanks provided to you.

See the web version of this resource for the environment diagram.

Video diagram

### Q10: Nested Calls Diagrams

Draw the environment diagram that results from executing the code below. You may not need to use all of the frames and blanks provided to you.

See the web version of this resource for the environment diagram.

Video walkthrough