Discussion 1: Environment Diagrams, Control

This is an online worksheet that you can work on during discussions and tutorials. Your work is not graded and you do not need to submit anything.

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.

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 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!

```
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 truth value of the following expression (so not will always return either `True` or `False`).
- `and` evaluates expressions in order and stops evaluating (short-circuits) once it reaches the first false value, and then 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
```

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 and returns `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
    """
    "*** YOUR CODE HERE ***"


```

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):
    "*** YOUR CODE HERE ***"


```

Q2: (Tutorial) Warm Up: Case Conundrum

These exercises are meant to help refresh your memory of topics covered in lecture and/or lab this week before tackling more challenging problems.

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

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?

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>:
    <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.

Q3: 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))
```

Q4: (Tutorial) 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
    """
    "*** YOUR CODE HERE ***"


```

Q5: (Tutorial) Fizzbuzz

Implement `fizzbuzz(n)`, which prints numbers from 1 to `n`. However, for numbers divisible by 3, print "fizz". For numbers divisible by 5, print "buzz". For numbers divisibule by both 3 and 5, print "fizzbuzz".

This is a standard software engineering interview question, but 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
    """
    "*** YOUR CODE HERE ***"

```

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 simple program and its corresponding diagram:

Server error! Your code might have an INFINITE LOOP or be running for too long.
The server may also be OVERLOADED. Or you're behind a FIREWALL that blocks access.
Try again later. This site is free with NO technical support. [#UnknownServerError]

(see UNSUPPORTED FEATURES)

Remember that programs are simply 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:

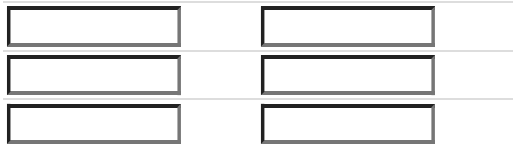
1. Evaluate the expression on the right side of the `=` sign
2. Write the variable name and the expression's value in the current frame.

Q6: Assignment Diagram

Use these rules to draw a simple diagram for the assignment statements below:

```
x = 11 % 4
y = x
x **= 2
```

Global frame



`def` Statements

A `def` statement creates 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. **Very important note:** Assignments for def statements use pointers to functions, which can have different behavior than primitive assignments.

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.

Q7: def Diagram

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

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

def triple(x):
    return x * 3

hat = double
double = triple
```

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`.

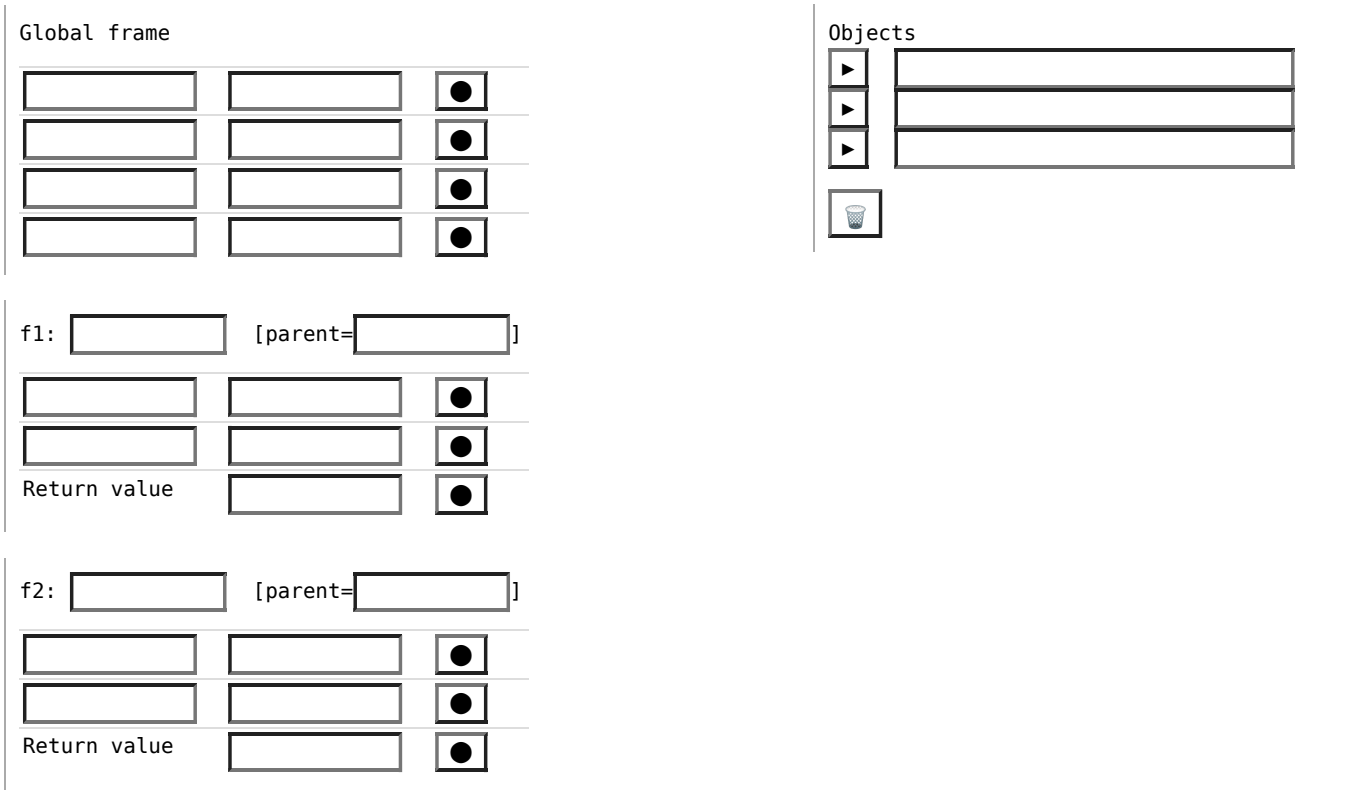
Note: Since we do not know how built-in functions like `min(...)` or imported functions like `add(...)` are implemented, we do not draw a new frame when we call them, since we would not be able to fill it out accurately.

Q8: 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.

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

hmmm = double
wow = double(3)
hmmm(wow)
```



Q9: (Tutorial) 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.

```
def f(x):
    return x

def g(x, y):
    if x(y):
        return not y
    return y

x = 3
x = g(f, x)
f = g(f, 0)
```