

Discussion 1: Control, Environment Diagrams

This is an online worksheet that you can work on during discussions. Your work is not graded and you do not need to submit anything. The last section of most worksheets is Exam Prep, which will typically only be taught by your TA if you are in an Exam Prep section. You are of course more than welcome to work on Exam Prep problems on your own.

Vitamin

The vitamin for Discussion 1 is worth 1 point and located here (<https://links.cs61a.org/vitamin1>).

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 `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!

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

First, try solving this problem using an if statement.

```
1  def wears_jacket_with_if(temp, raining):
2      """
3          >>> wears_jacket_with_if(90, False)
4          False
5          >>> wears_jacket_with_if(40, False)
6          True
7          >>> wears_jacket_with_if(100, True)
8          True
9          """
10     """ *** YOUR CODE HERE *** """
11
12
```

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.

```
1  def wears_jacket(temp, raining):
2      """ *** YOUR CODE HERE *** """
3
4
```

Q2: Case Conundrum

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: 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`.

```
1 def is_prime(n):
2     """
3     >>> is_prime(10)
4     False
5     >>> is_prime(7)
6     True
7     """
8     """ YOUR CODE HERE """
9
10
```


Q5: 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 divisible 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!

```
1  def fizzbuzz(n):
2      """
3      >>> result = fizzbuzz(16)
4      1
5      2
6      fizz
7      4
8      buzz
9      fizz
10     7
11     8
12     fizz
13     buzz
14     11
15     fizz
16     13
17     14
18     fizzbuzz
19     16
20     >>> result == None
21     True
22     """
23     """ *** YOUR CODE HERE *** """
24
25
```

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

<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>

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
```

Global frame

<input type="text"/>	<input type="text"/>	<input type="checkbox"/>
<input type="text"/>	<input type="text"/>	<input type="checkbox"/>
<input type="text"/>	<input type="text"/>	<input type="checkbox"/>

Objects

<input type="checkbox"/>	<input type="text"/>
<input type="checkbox"/>	<input type="text"/>
<input type="checkbox"/>	<input type="text"/>
<input type="checkbox"/>	

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)
```

Global frame

		●
		●
		●
		●

Objects

▶	
▶	
▶	
🗑️	

f1: [parent=]

		●
		●
Return value		●

f2: [parent=]

		●
		●
Return value		●

Q9: 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)
```

Global frame

		●
		●
		●

f1: [] [parent= []]

		●
		●
Return value		●

f2: [] [parent= []]

		●
		●
Return value		●

f3: [] [parent= []]

		●
		●
Return value		●

f4: [] [parent= []]

		●
		●
Return value		●

Objects

▶	
▶	
▶	
🗑	

Exam prep

We recommending reading sections 1.1-1.5 from the textbook for these problems.

Q10: Unique Digits

Write a function that returns the number of unique digits in a positive integer.

Hint: You may find it helpful to first define a function `has_digit(n, k)`, which determines whether a number `n` has digit `k`.

```
1 def unique_digits(n):
2     """Return the number of unique digits in positive integer n
3
4     >>> unique_digits(8675309) # All are unique
5     7
6     >>> unique_digits(1313131) # 1 and 3
7     2
8     >>> unique_digits(13173131) # 1, 3, and 7
9     3
10    >>> unique_digits(10000) # 0 and 1
11    2
12    >>> unique_digits(101) # 0 and 1
13    2
14    >>> unique_digits(10) # 0 and 1
15    2
16    """
17    """ YOUR CODE HERE """
18
19
```

We have provided two solutions:

- In one solution, we look at the current digit, and check if the rest of the number contains that digit or not. We only say it's unique if the digit doesn't exist in the rest. We do this for every digit.
- In the other, we loop through the numbers 0-9 and just call `has_digit` on each one. If it returns true then we know the entire number contains that digit and we can one to our unique count.

Video walkthrough:

Q11: Run, K, Run

An *increasing run* of an integer is a sequence of consecutive digits in which each digit is larger than the last. For example, the number 123444345 has four increasing runs: 1234, 4, 4 and 345. Each run can be indexed **from the end** of the number, starting with index 0. In the previous example, the 0th run is 345, the first run is 4, the second run is 4 and the third run is 1234.

Implement `get_k_run_starter`, which takes in integers `n` and `k` and returns the 0th digit of the `k`th increasing run within `n`. The 0th digit is the leftmost number in the run. You may assume that there are at least `k+1` increasing runs in `n`.

```

1  def get_k_run_starter(n, k):
2      """
3      >>> get_k_run_starter(123444345, 0) # example from description
4      3
5      >>> get_k_run_starter(123444345, 1)
6      4
7      >>> get_k_run_starter(123444345, 2)
8      4
9      >>> get_k_run_starter(123444345, 3)
10     1
11     >>> get_k_run_starter(123412341234, 1)
12     1
13     >>> get_k_run_starter(1234234534564567, 0)
14     4
15     >>> get_k_run_starter(1234234534564567, 1)
16     3
17     >>> get_k_run_starter(1234234534564567, 2)
18     2
19     """
20     i = 0
21     final = None
22     while _____:
23         while _____:
24             _____
25             final = _____
26             i = _____
27             n = _____
28     return final
29 
```

Q12: Environment Diagrams - Challenge

These questions were originally developed by Albert Wu (<http://albertwu.org/cs61a/>) and are included here for extra practice. We recommend checking your work in PythonTutor (<https://tutor.cs61a.org>) after filling in the diagrams for the code below.

Challenge 1

Draw the environment diagram that results from executing the code below.

Guiding Notes:


- Pay special attention to the names of the frames!

```
def funny(joke):  
    hoax = joke + 1  
    return funny(hoax)  
  
def sad(joke):  
    hoax = joke - 1  
    return hoax + hoax  
  
funny, sad = sad, funny  
result = funny(sad(1))
```

Global frame

		●
		●
		●
		●
		●

Objects

▶	
▶	
▶	
▶	
▶	
	

f1: [parent=]

		●
		●
		●
Return value		●

f2: [parent=]

		●
		●
		●
Return value		●

f3: [parent=]

		●
		●
		●
Return value		●

Challenge 2

Draw the environment diagram that results from executing the code below.

```
def double(x):
    return double(x + x)

first = double

def double(y):
    return y + y

result = first(10)
```

Global frame

		●
		●
		●
		●
		●

f1: [parent=]

		●
		●
		●
Return value	<input type="text"/>	●

f2: [parent=]

		●
		●
		●
Return value	<input type="text"/>	●

f3: [parent=]

		●
		●
		●
Return value	<input type="text"/>	●

Objects

▶	<input type="text"/>
▶	<input type="text"/>
▶	<input type="text"/>
▶	<input type="text"/>
▶	<input type="text"/>
