CONTROL & ENVIRONMENTS

- python
  - ↳ False-y values are (False, 0, "", None, [], {})
  - ↳ everything else is a Truth-y value
- 'and', 'or' always return the last type of the thing it saw (take 2 arguments)

```
>> 1 and 4
   4
>> 3 or False
   3
>> 0 or False
   False
```

- 'not' returns either True or False (takes 1 argument)

```
>> not 1
   False
>> not False
   True
```

- short-circuiting → stop evaluating when condition is satisfied

```
>> True or 1/0
   True
>> False and 1/0
   False
```

} even though 1/0 usually causes an error, both of these are actually valid expressions because first value is enough to find correct evaluation
  - ↳ OR short-circuits to True @ first True element
  - ↳ AND short-circuits to False @ first False element

} INTUITION BUILDING: test out some examples to convince yourself that this is true!

- "Nested" Boolean operators

```
>> 1 and 3 or 4
```

← have to simplify argument 2 down to just a single #/boolean first

1 = argument 1, and = operator, 3 or 4 = argument 2

```
3 or 4
  3
```

} short-circuit @ first True element

```
1 and 3
```

1 = argument 1, and = operator, 3 = argument 2

```
  3
```

```
>> 5 and 'false' or 4 or 1/0
```

→ remember this is a truth-y value! (all strings except for empty string are true in python)

```
5 and 'false' or 4 or 1/0
      'false' or 4 or 1/0
                 4 or 1/0
                   4
```

← no error from 1/0 because of short-circuiting!

```
'false'
'false'
```

- while loops

```
while <expr>:
    <body>
```

→ while some expression 'expr' holds true, continuously execute 'body'
(watch out for infinite loops here! If 'expr' never becomes false, we will never leave the loop!)

- if-statements

can control what should happen based on certain conditions

```
if <condition1>:
    <do thing1>      ← only happens (we only enter here) if condition1 is true
elif <condition2>:   ← short for 'else if'
    <do thing2>
else:
    <do thing3>      ← only enter here if neither condition1 nor condition2 are true
```

- environment diagrams (more next week!)
  - ↳ help us keep track of & understand complicated code
  - ↳ use PYTHONTUTOR (online tool to convert code to its env. diagram)

key: open a new frame each time you call a function (start w/ global frame)
(always keep track of 'parent' frame → frame that called given function)

- expressions vs. statements
  - → expressions evaluate to values (may have side effects that modify state of program)
  - → statements modify the flow of program (don't evaluate to anything)

ex.)
```
>> 3+1
4
>> x=5
```

Expression: (3+1)=4 python evaluates 3+1 & outputs the value 4

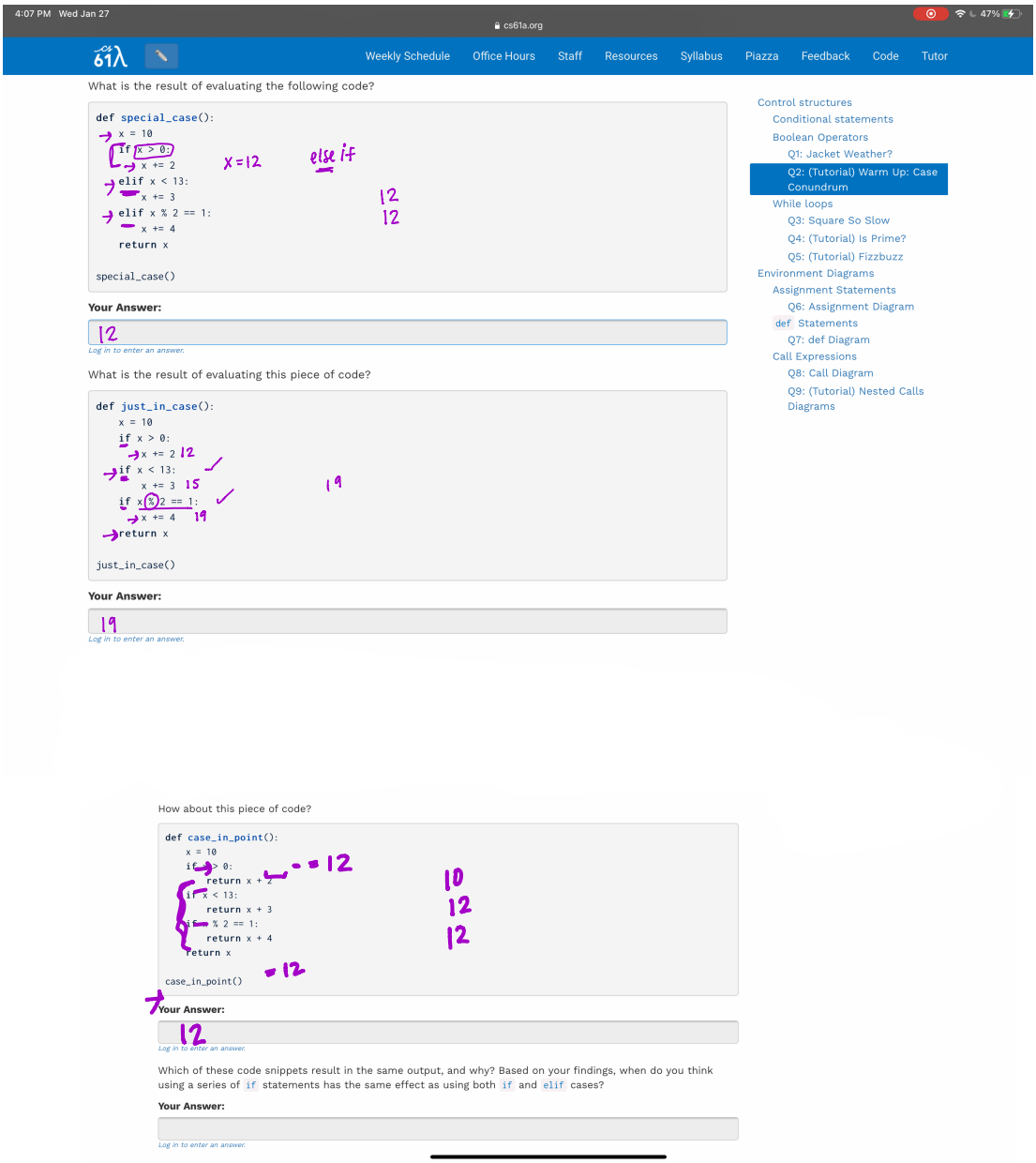
statement: binds the value 5 to the variable x so that later in the program if you write 'x' python knows you mean 5

- evaluation rules (will cover more w/ env. diagrams next week!)
  - Function calls
    1. evaluate operator
    2. evaluate operands (L→R)
    3. apply the operator to the values of operands

  ENV. DIAG.
  1. create new frame
  2. Bind formal params w/ actual argument vals
  3. evaluate body of function

  - Assignment
    1. evaluate expressions on RHS of equals sign (L→R)
    2. bind all names on LHS to resulting values
    3. names can only bind to values, not other names!
  - Lookup
    1. Lookup in current frame
    2. if not here, check parent frame, etc.

warm-up : Q2



4:07 PM Wed Jan 27 — cs61a.org — 47%

cs 61A | Weekly Schedule | Office Hours | Staff | Resources | Syllabus | Piazza | Feedback | Code | Tutor

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()
```

x = 12 else if 12 12

Your Answer:

12

Log in to enter an answer.

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()
```

12 15 19 19

Your Answer:

19

Log in to enter an answer.

Control structures
- Conditional statements
- Boolean Operators
  - Q1: Jacket Weather?
  - Q2: (Tutorial) Warm Up: Case Conundrum
- While loops
  - Q3: Square So Slow
  - Q4: (Tutorial) Is Prime?
  - Q5: (Tutorial) Fizzbuzz

Environment Diagrams
- Assignment Statements
  - Q6: Assignment Diagram
- `def` Statements
  - Q7: def Diagram
- Call Expressions
  - Q8: Call Diagram
  - Q9: (Tutorial) Nested Calls Diagrams

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()
```

= 12 10 12 12 = 12

Your Answer:

12

Log in to enter an answer.

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?

Your Answer:

Log in to enter an answer.

Practice Problems from Discussion Wksht

Q4: Is Prime?

1.3 **Tutorial:** 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
    """
```

always annotate problem to make sure you don't miss important parts!

1.3 **Tutorial:** 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`.

we know n is positive integer: {1, 2, 3, ...}

oftentimes problem description gives good idea/approach

another tool we can see will be helpful in solving problem

```
def is_prime(n):
    """
    >>> is_prime(10)
    False
    >>> is_prime(7)
    True
    """
```

always look @ doctests to spot edge cases!

10 is not prime because it is divisible by more than just 1 and 10

7 is prime because it is only divisible by 1 and 7

Pseudocode (explicitly say what we want to do in words):

Let input to function be n (we want to check if n is prime).

check if n is divisible by 1 and itself.

↳ In reality, do we really need this check? Is it always true or always false?

If n is divisible by any other integer (not 1 or n), conclude n is prime.

otherwise, n is not prime.

{2, ..., n-1}

available tools:

x % y (mod operator) → remainder when x is divided by y
↳ how can we use this to check the divisibility of #s?
↳ what should x%y equal when x is divisible by y? 0

$n \in \{1, 2, \dots\}$

```
def is_prime(n):
    For all #s K in {2,3,..., n-1}:
        If n is divisible by K:
            return False
    return True
```

← we only reach here after we have successfully checked that all 'K' are not factors of n (ie. only 1 & n are factors of n)

Ⓔ go back to original problem; are we missing smthg? Is there a possible input n that we are not accounting for? Ⓔ

↳ YES!! we said all possible inputs are positive integers.
↳ 1 is a positive integer & we know 1 is not a prime number
But, what does the code return as of right now?

```
def is_prime(n):
    if n==1:
        return False
    K=2
    while K<n:
        if n%K==0:
            return False
        K = K+1
    return True
```

} special case that doesn't fall into our general algorithm (follows from orange above)

] For all #s from 2 to n-1

→ check if n is divisible by given #

] increment K by one so that we are checking $\{2, \dots, n-1\}$

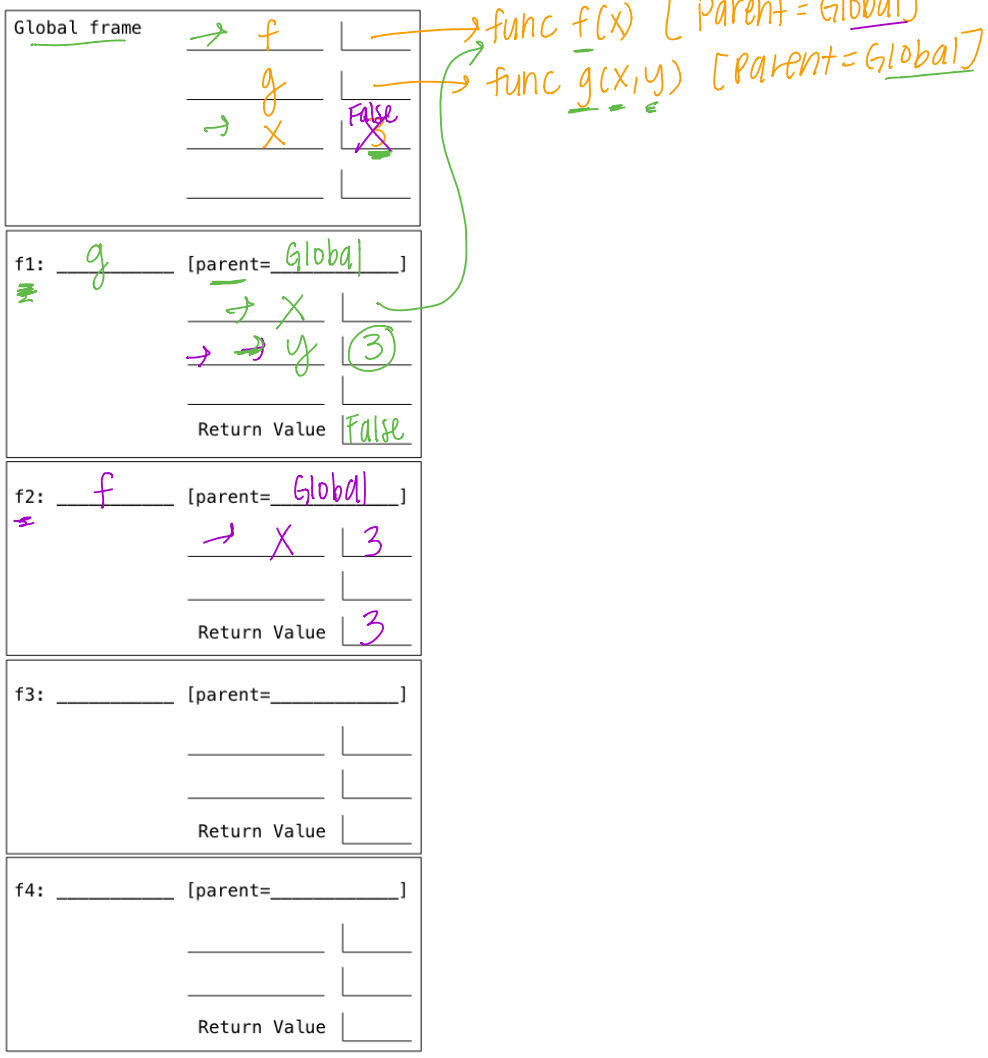
Q9: Nested Call Diagrams

2.4 **Tutorial:** Draw the environment diagram that results from executing the code below.

```
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)
```



2.4 **Tutorial:** Draw the environment diagram that results from executing the code below.

```
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)
```

f(3) = 3

f(0) = 0