FINAL REVIEW SESSION!

Presented By: Ian, Lenhard, Riya, Cornelius, Derek, and Tiffany

TOPICS FOR THE EXAM

IN THIS PRESENTATION

- Recursion
- For Loops
- While Loops
- String Slicing
- Testing/Debugging
- Searching/Sorting

ON THE EXAM BUT NOT IN THIS PRESENTATION

- Call Frames
- Classes + Subclasses + Inheritance

STRING PROCESSING

STRING PROCESSING

- Use common string methods to also get pieces of a string
- Common methods are shown in a few slides
- We can also slice strings with bracket!
- What would x = s[3:] give us?

s =

0	1	2	3	4	5	6
A	S	T	R	I	N	G

STRING PROCESSING

- Use common string methods to also get pieces of a string
- Common methods are shown in a few slides
- We can also slice strings with bracket!
- What would x = s[3:] give us?

s =

0	1	2	3	4	5	6
A	S	T	R	I	N	G

STRING PROCESSING

- What would x = s[3:] give us?

s =

0	1	2	3	4	5	6
A	S	T	R	I	N	G

x =

0	1	2	3
R	I	N	G

REMEMBER WITH STRING SLICING...

```
>>> s[start_pos:end_pos]
```

- This makes a new string, so return it or store it in a variable!
- If you leave start_pos out, Python "fills it in" with 0

  `>>> s[:end_pos]` is the same as `>>> s[0:end_pos]`

- If you leave end_pos out, Python "fill it in" with len(s)

  `>>> s[start_pos:]` is the same as `>>> s[start_pos:len(s)]`

- The ending index is "non-inclusive," so Python DOES NOT include it in the new string!

OTHER IMPORTANT STRING METHODS

- string.index(substring)
  - ⑩ Returns the first occurrence of substring inside of string
  - ⑩ Gives error if substring is not in string
- ⑩ string.find(substring)
  - ⑩ Returns the first occurrence of substring inside of string
  - ⑩ Returns –1 if substring is not in string
- ⑩ string.rfind(substring)
  - ⑩ Returns the LAST occurrence of substring inside of string
  - ⑩ Returns –1 if substring is not in string
- ⑩ string.strip()
  - ⑩ Returns a copy of string with white-space removed at ends

Remember:

We call string METHODS with

string.method_name(args)

String name goes in front!

LET'S PRACTICE!

```
def secret_message(msg):
    """
    Returns the secret messages inside of the msg.

    The secret message in msg will be the string after the first occurence of
    '<<' but before the last occurence of '>>' with white any potential leading
    and trailing spaces removed. There could be an unknown number of
    spaces between '<<', the message, and '>>'.

    For example,
        secret_message("urie>>anveovn<<      You're awesome!!!   >>vudbku<<")
    returns:
        'You're awesome!!!' (notice, leading and trailing spaces removed)

    And,
        secret_message("<<U R Cool >> >> >>")
    returns:
        'U R Cool >> >>' (notice, the message ends at the LAST occurence of '>>')

    Precondition: msg is a string with at least one occurence of '<<' before
        the last occurence of '>>'
    """
```

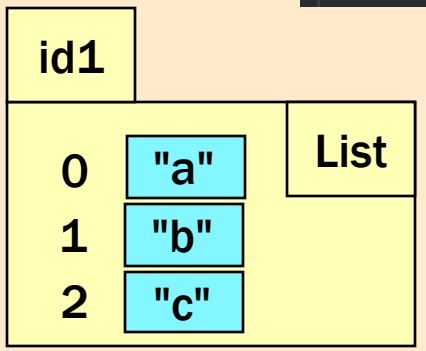
FOR LOOPS

QUICK REVIEW: DICTIONARIES AND LISTS

Lists

- Used to store multiple values in the same variable.
- Each item is "labeled" with an index, ranging 0 to len(list) - 1
  - Access elements with these indices

```
tlist = ["a","b","c"]
```



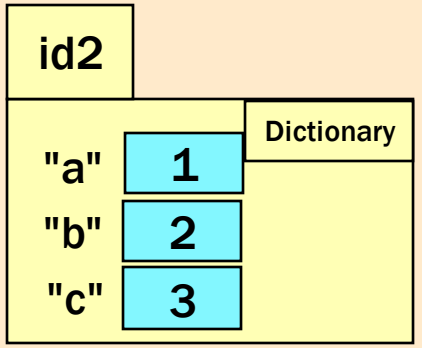
Use tlist[0] to access "a"

Dictionaries

- Used to store multiple values in the same variable.
- Each value is "labeled" with a key and items are stored in "key-value pairs"
  - Access elements with these keys

```
dict = {"a":1,"b":2,"c":3}
```

Use dict["a"] to access 1



FOR LOOPS (STRUCTURE)

Usually a new variable. Python creates this in increments automatically

A list, a dictionary, a string, or anything else iterable.

```
for <loop_variable> in <something_to_loop_over>:
    <do something>
```

More on what "something" can be in the next slide

Remember, if you return in a for loop, this stops the loop early (sometimes this is intended; sometimes it is not)

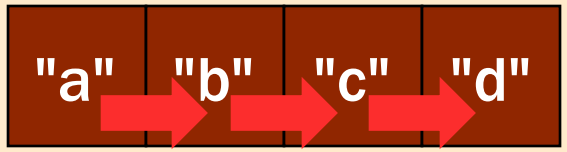
FOR LOOPS (TYPES)

Looping Over Items

```
alist = ["a","b","c","d"]
for item in alist:
    <do_something>
```

- This loop is good for processing items of a list:
  - Looking at items, storing them elsewhere, etc.
- These are NOT good for editing lists

Values of item on each iteration:

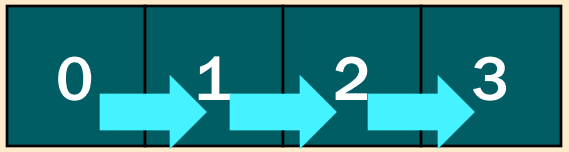


Looping Over Indices

```
alist = ["a","b","c","d"]
for pos in range(len(alist)):
    <do_something>
```

- This loop is good for editing items of a list or looking a positions.
- The indices (pos) can be used to change the value of an entry in the list.

Values of pos on each iteration:



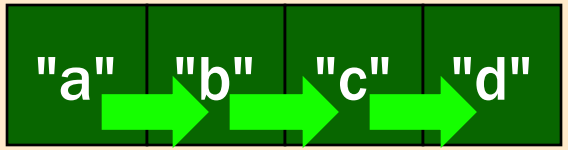
FOR LOOPS (TYPES)

Looping Over Dictionaries

```
dict = {"a":1,"b":2,"c":3}
for key in dict:
    <do_something>
```

- This loop is good for processing AND editing items of a dictionary:
- During each iteration, key holds the value of a key for the dictionary.
- We can use this to get values of dictionary and edit values of dictionary

Values of key on each iteration:



Pro-Tips from CS 1110 Staff

It sometimes helps to give the loop variable a name that tells you what it is (instead of just x or y), like

pos or idx for range-len loops

item for regular for loops

key for dictionary loops

LET'S PRACTICE!

```
def max_cols(table):
    """
    Returns: a list that contains the max value of each column

    For example:
    If table = [[1, 2, 3],
                [4, 5, 6],
                [7, 3, 9]]
    Then max_cols(A) returns [7, 5, 9]

    Precondition: table is a NONEMPTY 2D list of floats
    """
```

SOLUTION

```
lst = []
for col in range(len(table[0])):
    max_element = table[0][col]
    for row in range(len(table)):
        if max_element < table[row][col]:
            max_element = table[row][col]
    lst.append(max_element)
return lst
```

WHILE LOOPS

WHILE LOOPS

- Basically, while a condition is true, do something
- Do something an unknown number of times
- More freedom than for loops (Python does not make/increment a loop variable)
- But, this can lead to more bugs

⑩ Make sure you ALWAYS ensure that your code makes progress towards making that condition false or your while loop will go on forever…

```
while <condition >:
    statement 1
    …
    statement n
```

body

WHILE LOOPS (EXAMPLES)

This must be a Boolean (or Boolean expression)

```
while <condition >:
    statement 1
    …
    statement n
```

body

Must eventually do something here to <condition> is false at some point.

Unlike for loops, in while loops, Python does not make and increment this "loop variable" for us. So, we need to do this ourselves.

```
k = 0
while k < n:
    # do something
    k = k+1
```

Forgetting this means Python loops forever

LET'S PRACTICE!

```
def duplicate_again(nums,a):
    """
    MODIFIES `nums` so that all occurrences of characters that are not `a` are
    duplicated (where each duplicate is adjacent to the original.)
    Does not return anything.

    Examples:
    If nums1 = [1,2,3,1], then after duplicate(nums1,1),
            the list that nums1 stores is *altered* to [1, 2, 2, 3, 3, 1].

    If nums2 = [1,2,3,1], then after duplicate(nums2, 4), nums2 is altered to
            [1, 1, 2 ,2, 3, 3, 1, 1]

    Preconditions:
        nums: list of ints
        a: an int
    """
```

RECURSION

RECURSION (GENERAL IDEA)

- Technique that solve problems by breaking them down into sub-problems.

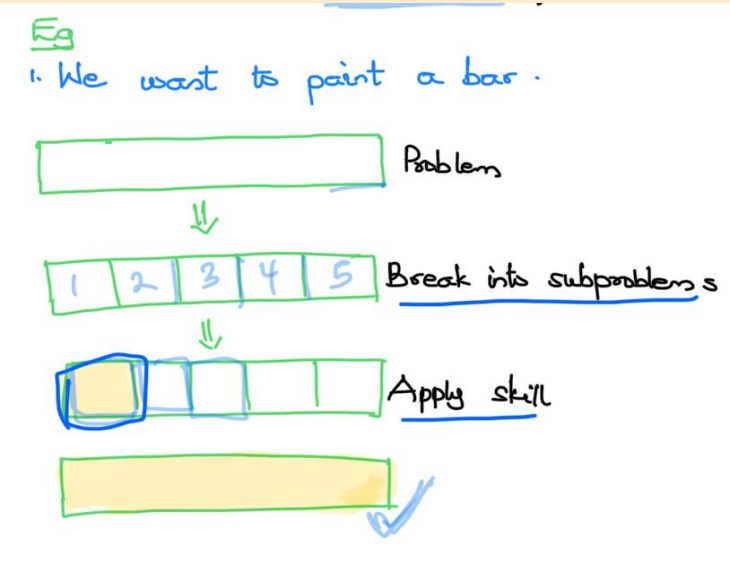
Uses a recursive function- a function that makes a call to itself during Its execution.

A recursive function has a **base case** and a **recursive case.**

Splitting:

- If a string / list, you can take the first element vs the remaining.

- If an object, take that object vs the target fields. (eg. children, employees)



WRITING A RECURSIVE FUNCTION:

Always READ the specification and do well to understand it.

Assume the function has been correctly implemented.

Step 1: Base case

- Is there a clear base case? If yes, implement it!

Step 2: Recursive Case

- Build-up on the cases and smaller recursive cases.
- Again: assume function is correctly implemented!

Step 3: Combine these outputs

- Usually the most challenging but understanding what the function does is extremely crucial.

RECURSION (TIPS)

Thinking of types sometimes helps (debugging and coding)

- If we know we need recursion and our function takes lists, this means we need to split our input into smaller lists
- If our function takes strings, this means we need to split our input into smaller strings

```
def num_es(s):
    """Returns: # of 'e's in s"""
```

This was bejeweled example from lecture!

- If our function takes Person objects, we need to split our input into "smaller" Person objects (perhaps with less ancestors?)

```
def find_waldo_broken(p):
    """  Returns:
             True  if any ancestor of p (including p) has the name "Waldo"
             False if no ancestor of p (including p) has the name "Waldo"
         Precondition (no need to assert): p is a person
    """
```

LET'S PRACTICE

1. Implement a function that adds all numbers in a list. List may contain nested lists.

2. Implement a function that counts vowels in a string.

3. Count number of dolls. Each doll may contain 0 or more children dolls.

LET'S PRACTICE!

6. [16 points] **Recursion.**

Let Employee be a class whose objects have the following two attributes:

```
name [str] - unique non-empty name of employee
employees [list of Employee] - employees reporting directly to this employee.
   *** LENGTH IS AT MOST 2 ***. (The length can be 0.)
```

Implement the following **function** (*not* a method), making effective use of recursion. For-loops are not required, although they are allowed as long as your solution is fundamentally recursive.

```
def get_tree(person):
    """Returns list of names of ALL employees that are *underneath*
    `person` (at any level). Order of list items does not matter; duplicates OK.

    Example:
    engineer = Employee('E', [])       means get_tree(engineer) --> []
    cto = Employee('T', [engineer])    means get_tree(cto) --> ['E']
    coo = Employee('O', [])            means get_tree(coo) --> []
    ceo = Employee('CEO', [coo, cto]) means get_tree(cea) --> ['O','T','E']

    Precondition: `person` is an Employee."""
```

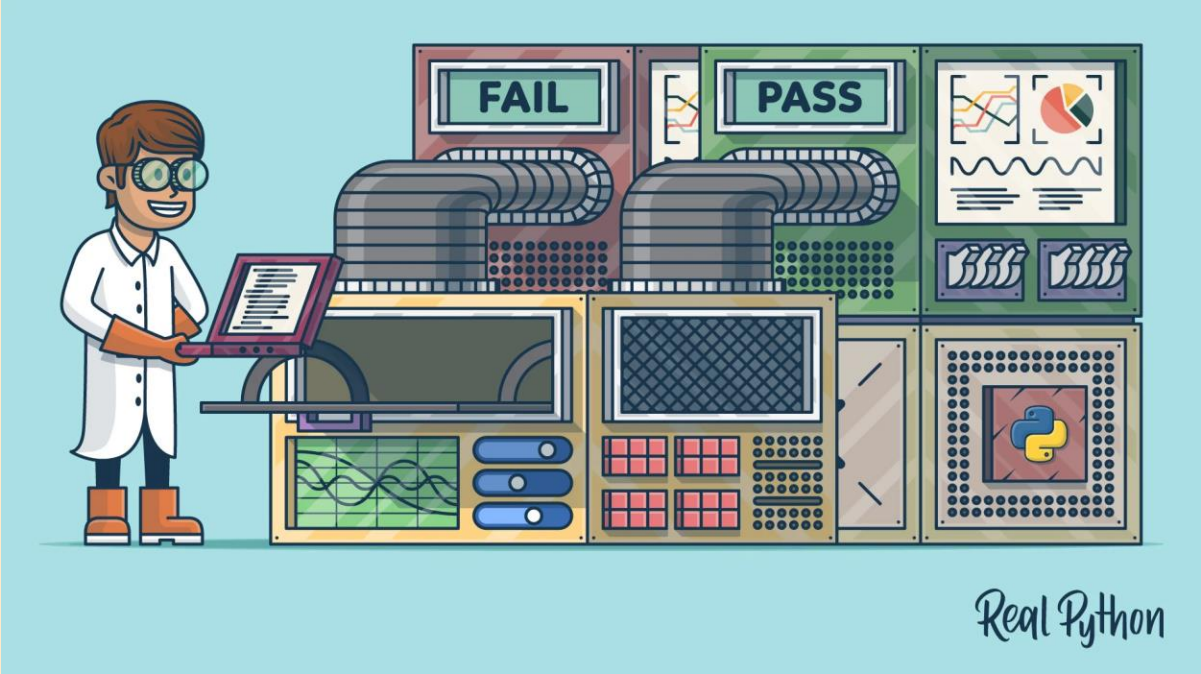
RECURSION (ALWAYS REMEMBER)

- Always call your function on "simpler" inputs
  - Make progress towards base case
  - Avoid infinite recursion
- Always make sure your code handles (explicitly or implicitly) the base case
- Always respect preconditions (only call function on valid inputs)
- Do these things to avoid Angry Python

TESTING

TESTING

- We can't test every input
- But, we can come up with "representative tests" that each have a significantly different input.
  - ❿ On the exam, tests will need to be significantly different to get credit
  - ❿ Be prepared to explain why your tests are different
- Sometimes we use the Rules of 0, 1, and Many to guide us



TESTING

Rule of 0

Test function on "0" (or empty) occurrences

- Empty string/list
- No occurrences of thing we looking for in the function

Rule of 1

Test function on "1" occurrence

- String/list of length 1
- Input has 1 occurrence of thing we are looking for

Rule of Many

Test function on "many" occurrences

- String/list with length greater than 1
- Multiple occurrences of the thing we are looking for.

REMEMBER: Tests must ALWAYS follow specification. If test does not follow specification, it does not count!

So, no empty list/string unless spec allows it

LET'S PRACTICE

- Let's make testing cases for the function from the last section!
- Recall the spec:

```
def duplicate_again(nums,a):
    """
    MODIFIES `nums` so that all occurrences of characters that are not `a` are
    duplicated (where each duplicate is adjacent to the original.)
    Does not return anything.

    Examples:
    If nums1 = [1,2,3,1], then after duplicate(nums1,1),
        the list that nums1 stores is *altered* to [1, 2, 2, 3, 3, 1].

    If nums2 = [1,2,3,1], then after duplicate(nums2, 4), nums2 is altered to
        [1, 1, 2 ,2, 3, 3, 1, 1]

    Preconditions:
      nums: list of ints
      a: an int
    """
```

LET'S PRACTICE

Rule of 0 | Rule of 1 | Rule of Many

- What can we use here? Is the empty list allowed?

LET'S PRACTICE

Rule of 0

- What can we use here? Is the empty list allowed?
  - ❿ Yes! So, one testing case will be:

Inputs	Output
nums = [], a = 1	[]

- Can we have a case where we don't repeat anything?

Rule of 1

Rule of Many

LET'S PRACTICE

Rule of 0

- What can we use here? Is the empty list allowed?

  ⑩ Yes! So, one testing case will be:

Inputs	Output
nums = [], a = 1	[]

- Can we have a case where we don't repeat anything?

  ⑩Yes! If we only have 'a' in our list! So, another testing case is:

Inputs	Output
nums = [1], a = 1	[1]

Rule of 1

- Can we have a test case where we only repeat 1 thing?

Rule of Many

LET'S PRACTICE

Rule of 0

- What can we use here? Is the empty list allowed?
  - ⑩ Yes! So, one testing case will be:

Inputs	Output
nums = [], a = 1	[]

- Can we have a case where we don't repeat anything?
  - ⑩Yes! If we only have 'a' in our list! So, another testing case is:

Inputs	Output
nums = [1], a = 1	[1]

Rule of 1

- Can we have a test case where we only repeat 1 thing?
  - Yes! If there is only 1 occurrence of a non-a element in nums, then only one thing will be repeated. So, another test case is:

Inputs	Output
nums = [1,2,1], a = 1	[1,2,2,1]

Rule of Many

- Can we have a test case where more than one thing is repeated in nums?

LET'S PRACTICE

Rule of 0

- What can we use here? Is the empty list allowed?
  - ⑩ Yes! So, one testing case will be:

Inputs	Output
nums = [], a = 1	[]

- Can we have a case where we don't repeat anything?
  - ⑩Yes! If we only have 'a' in our list! So, another testing case is:

Inputs	Output
nums = [1], a = 1	[1]

Rule of 1

- Can we have a test case where we only repeat 1 thing?
  - Yes! If there is only 1 occurrence of a non-a element in nums, then only one thing will be repeated. So, another test case is:

Inputs	Output
nums = [1,2,1], a = 1	[1,2,2,1]

Rule of Many

- Can we have a test case where more than one thing is repeated in nums?
  - Yes! We could have a test case where nums only contains non-a elements. So, one case is:

Inputs	Output
nums = [2,3,4], a = 1	[2,2,3,3,4,4]

- Is this actually distinct?

LET'S PRACTICE

Rule of 0

- What can we use here? Is the empty list allowed?
  - ⑩ Yes! So, one testing case will be:

Inputs	Output
nums = [], a = 1	[]

- Can we have a case where we don't repeat anything?
  - ⑩Yes! If we only have 'a' in our list! So, another testing case is:

Inputs	Output
nums = [1], a = 1	[1]

Rule of 1

- Can we have a test case where we only repeat 1 thing?
  - Yes! If there is only 1 occurrence of a non-a element in nums, then only one thing will be repeated. So, another test case is:

Inputs	Output
nums = [1,2,1], a = 1	[1,2,2,1]

Rule of Many

- Can we have a test case where more than one thing is repeated in nums?
  - Yes! We could have a test case where nums only contains non-a elements. So, one case is:

Inputs	Output
nums = [2,3,4], a = 1	[2,2,3,3,4,4]

- Is this actually distinct?
  - Yes! We need to check that our code actually looks for ALL non-a elements.

LET'S PRACTICE

So, our tests were:

Inputs	Output
nums = [], a = 1	[]

Inputs	Output
nums = [1], a = 1	[1]

Inputs	Output
nums = [1,2,1], a = 1	[1,2,2,1]

Inputs	Output
nums = [2,3,4], a = 1	[2,2,3,3,4,4]

Can you think of any more?

DEBUGGING

DEBUGGING

- Often hard to do
- Really tests your ability to "step through" code.
- Sometimes it helps to do pseudo-call-frames (informally, of course) to help map out what the method does on certain inputs.
- For recursion, still assume that methods works as intended whenever you call it
  - This will help find bugs in base case and combination steps
  - Still make sure the function is being called correctly though (like on proper inputs, etc.)

DEBUGGING (BRUNO...AGAIN)

- Here is a buggy implementation of earliest_bruno()
- There is one bug. What is it?

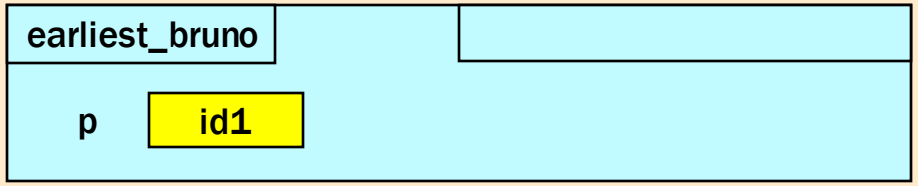
```python
def earliest_bruno(p):
    """
    Returns: the birthyear of the earliest born ancestor named "Bruno"
             None if there is no ancestor named "Bruno"
             this includes p

    Example: if there are two ancestors named "Bruno" born in 2000 and 1909,
                --> returns 1909

    Precondition (no need to assert): p is a person
    """
    bruno_births = []
    if p.name == "Bruno":
        bruno_births.append(p.birthyear)
    for parent in p.parents:
        bruno_birth_from_parent = earliest_bruno(parent)
        bruno_births.append(bruno_birth_from_parent)
    if bruno_births == []:
        return None
    else:
        return sorted(bruno_births)[0]
```

DEBUGGING

- Let's start testing inputs.
- What if id1.name is not "Bruno", id1.parents consists of p1 and p2, where p2 has an ancestor named "Bruno" but p1 does not



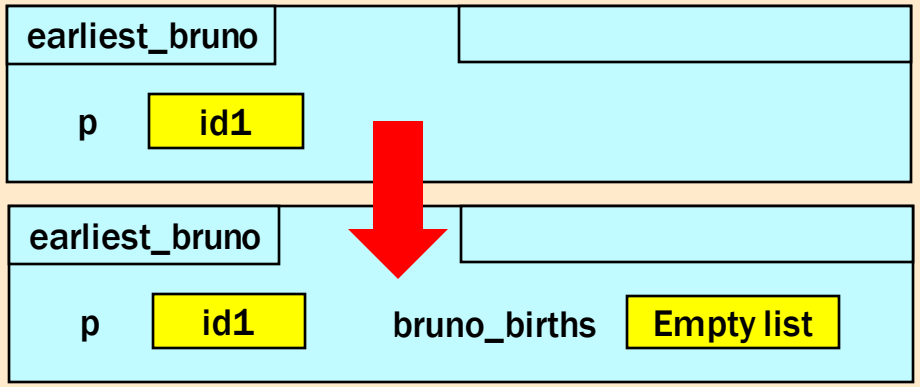

```python
def earliest_bruno(p):
    """
    Spec removed
    """
    bruno_births = []
    if p.name == "Bruno":
        bruno_births.append(p.birthyear)
    for parent in p.parents:
        bruno_birth_from_parent = earliest_bruno(parent)
        bruno_births.append(bruno_birth_from_parent)
    if bruno_births == []:
        return None
    else:
        return sorted(bruno_births)[0]
```

These drawings are not valid call frames!

DEBUGGING

- What if id1.name is not "Bruno", id1.parents consists of p1 and p2, where p2 has an ancestor named "Bruno" but p1 does not

```
def earliest_bruno(p):
    """
    Spec removed
    """
    bruno_births = []
    if p.name == "Bruno":
        bruno_births.append(p.birthyear)
    for parent in p.parents:
        bruno_birth_from_parent = earliest_bruno(parent)
        bruno_births.append(bruno_birth_from_parent)
    if bruno_births == []:
        return None
    else:
        return sorted(bruno_births)[0]
```

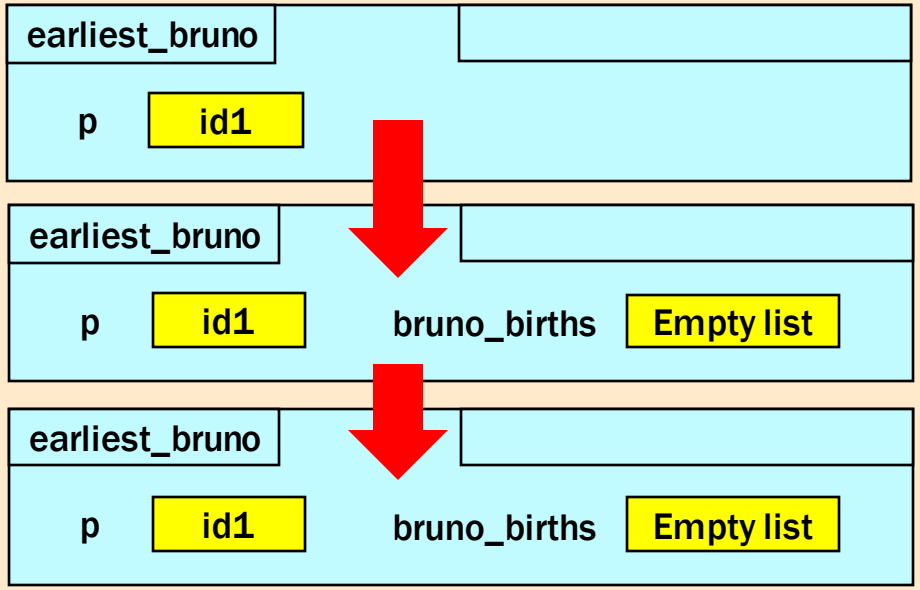


DEBUGGING

- What if id1.name is not "Bruno", id1.parents consists of p1 and p2, where p2 has an ancestor named "Bruno" but p1 does not

```python
def earliest_bruno(p):
    """
    Spec removed
    """
    bruno_births = []
    if p.name == "Bruno":
        bruno_births.append(p.birthyear)
    for parent in p.parents:
        bruno_birth_from_parent = earliest_bruno(parent)
        bruno_births.append(bruno_birth_from_parent)
    if bruno_births == []:
        return None
    else:
        return sorted(bruno_births)[0]
```

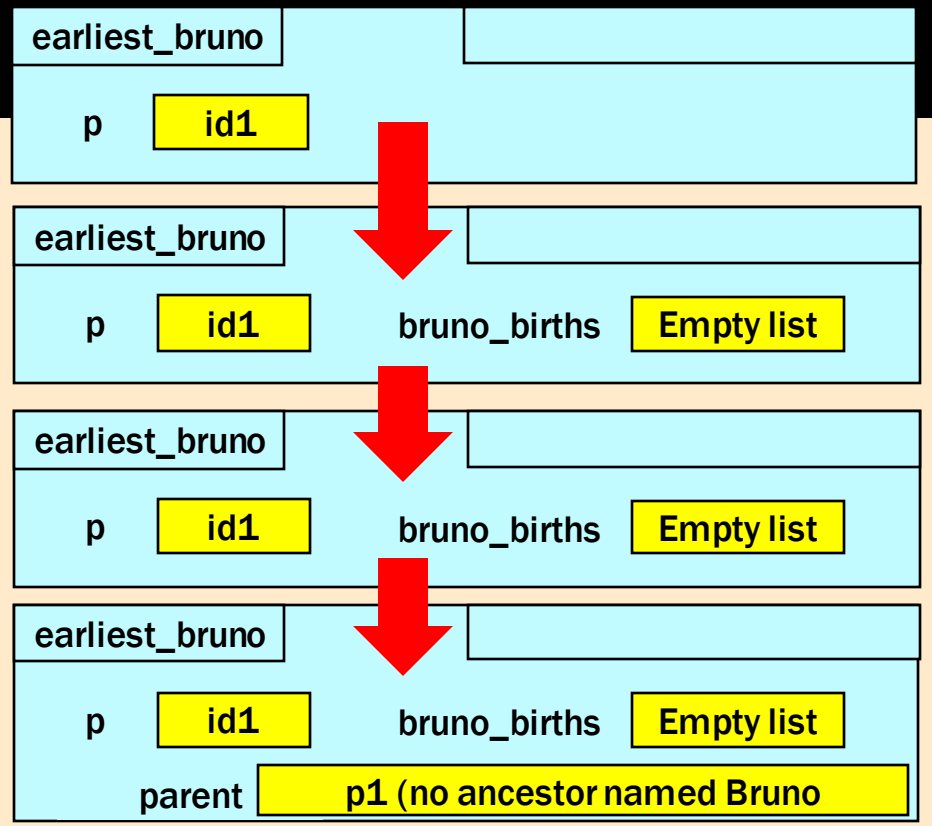
id1.name is not Bruno, so skip this



DEBUGGING

- What if id1.name is not "Bruno", id1.parents consists of p1 and p2, where p2 has an ancestor named "Bruno" but p1 does not

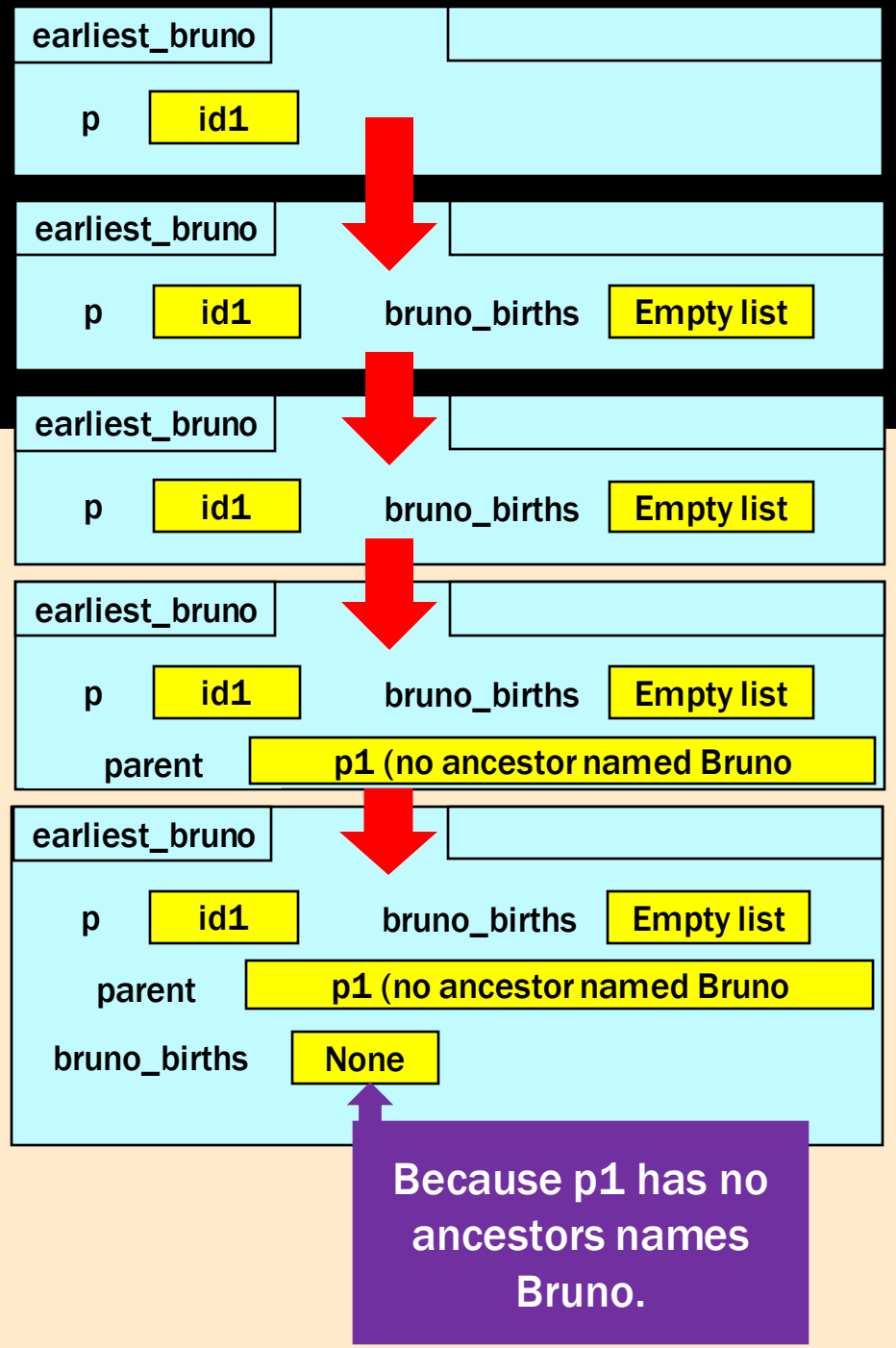
```python
def earliest_bruno(p):
    """
    Spec removed
    """
    bruno_births = []
    if p.name == "Bruno":
        bruno_births.append(p.birthyear)
    for parent in p.parents:
        bruno_birth_from_parent = earliest_bruno(parent)
        bruno_births.append(bruno_birth_from_parent)
    if bruno_births == []:
        return None
    else:
        return sorted(bruno_births)[0]
```



DEBUGGING

- What if id1.name is not "Bruno", id1.parents consists of p1 and p2, where p2 has an ancestor named "Bruno" but p1 does not

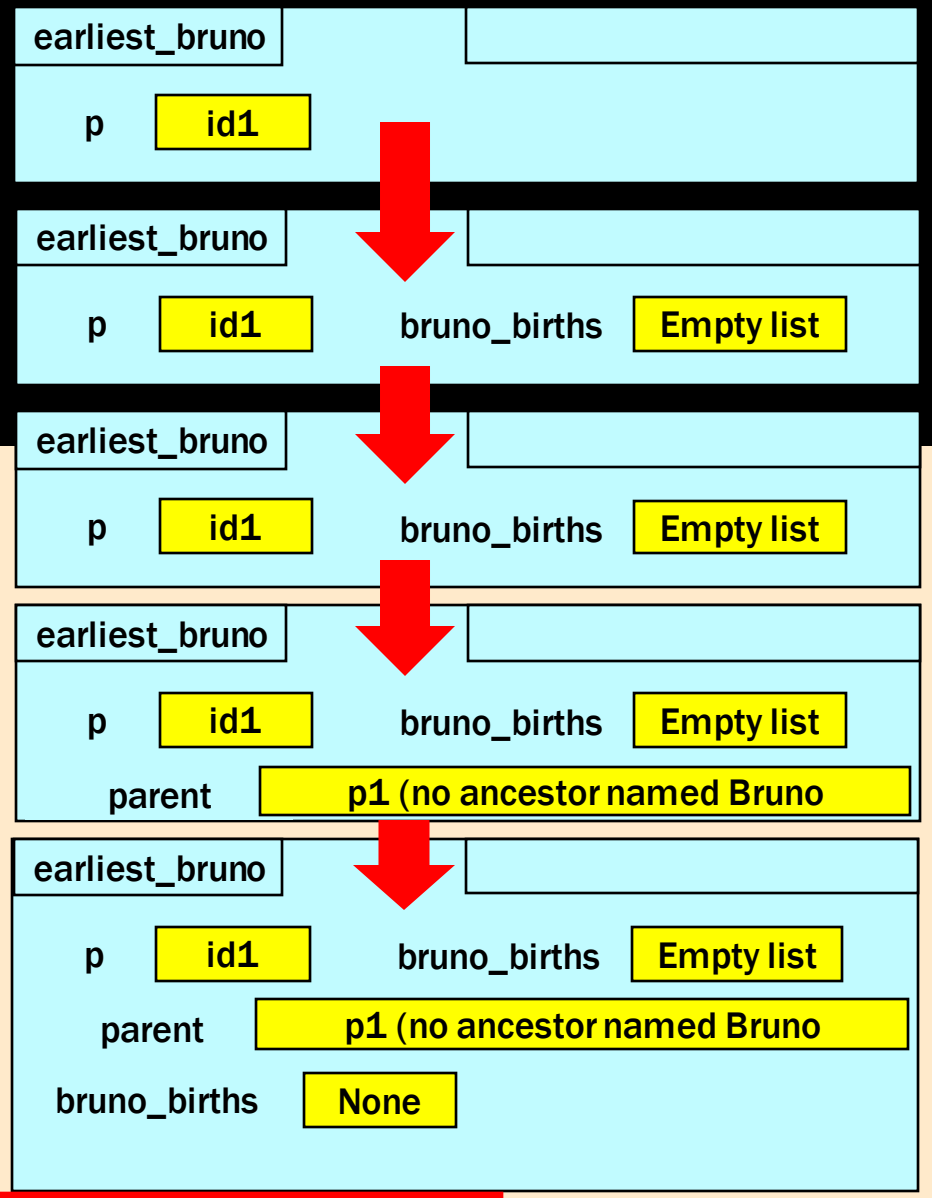
```
def earliest_bruno(p):
    """
    Spec removed
    """
    bruno_births = []
    if p.name == "Bruno":
        bruno_births.append(p.birthyear)
    for parent in p.parents:
        bruno_birth_from_parent = earliest_bruno(parent)
        bruno_births.append(bruno_birth_from_parent)
    if bruno_births == []:
        return None
    else:
        return sorted(bruno_births)[0]
```



DEBUGGING

- What if id1.name is not "Bruno", id1.parents consists of p1 and p2, where p2 has an ancestor named "Bruno" but p1 does not

```
def earliest_bruno(p):
    """
    Spec removed
    """
    bruno_births = []
    if p.name == "Bruno":
        bruno_births.append(p.birthyear)
    for parent in p.parents:
        bruno_birth_from_parent = earliest_bruno(parent)
        bruno_births.append(bruno_birth_from_parent)
    if bruno_births == []:
        return None
    else:
        return sorted(bruno_births)[0]
```

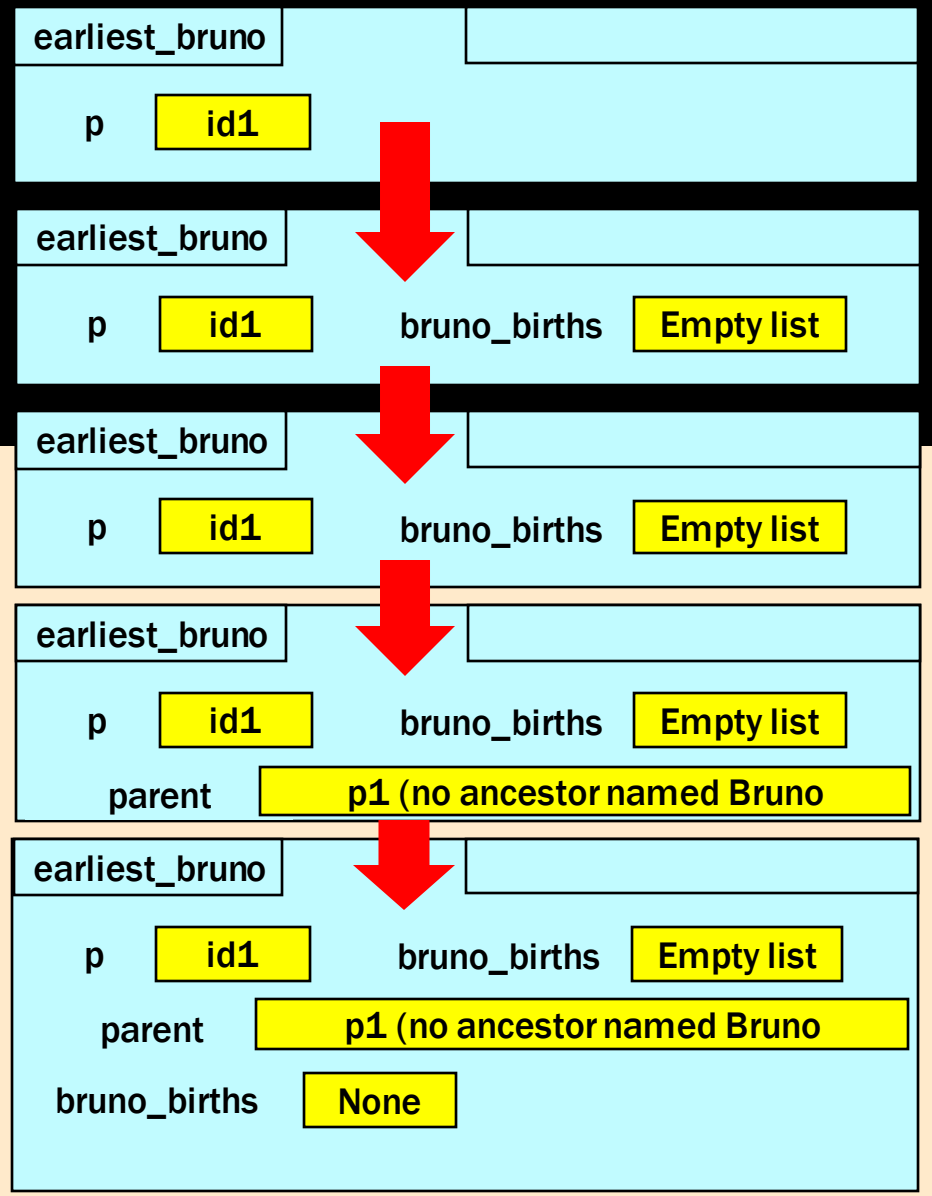


But, wait, this line will then append None to bruno_births. Is that what we want?

DEBUGGING

- What if id1.name is not "Bruno", id1.parents consists of p1 and p2, where p2 has an ancestor named "Bruno" but p1 does not

```
def earliest_bruno(p):
    """
    Spec removed
    """
    bruno_births = []
    if p.name == "Bruno":
        bruno_births.append(p.birthyear)
    for parent in p.parents:
        bruno_birth_from_parent = earliest_bruno(parent)
        bruno_births.append(bruno_birth_from_parent)
    if bruno_births == []:
        return None
    else:
        return sorted(bruno_births)[0]
```



No! Because this line tries to sort bruno_years. Python can't do that if there are Nones in a list with ints. So, we found our bug.

SORTING/SEARCHING

LINEAR SEARCH

- Input can be any iterable
  - Iterables are types that can be looped over (e.g. strings and lists)
- Iterate from start to end in search of the value
  - Can also search from end to start
- Time complexity is order of n, which we can write as O(n), where n is the length of the input
  - Worst case, value is not in list and algorithm searches entire list

BINARY SEARCH

- Input must be a sorted iterable
  - A string of letters in alphabetical order works
  - A string of letters and numbers does not work
  - A list of words in alphabetical order works
- Time complexity is order of logn, which we can write as O(logn), where n is the length of the input
  - Process of splitting in half is logarithmic, and this is done n times

INSERTION SORT

- Input iterable becomes slightly more sorted per iteration
  - Starting at the 2nd element, push values down to correct spot
  - E.g. [3,1,2] alters to [1,3,2] after an iteration since the '1' was pushed down to before '3'
  - Pushing elements down can be done in a helper method
- Time complexity is order of n^2, which we can write as O(n^2), where n is the length of the input
  - For every element in the array, we push the value down to its correct spot

MERGE SORT

- Perfect example of the power of recursion
  - Break input into two parts
  - Sort the two parts
  - Merge the two sorted lists into one list
- Time complexity is order of nlogn, which we can write as O(nlogn), where n is the length of the input
  - For every element in the array, we push the value down to its correct spot

THANK YOU FOR COMING AND GOOD LUCK

Just know, you got this!

Read the directions carefully

Breathe

And avoid Angry Python at all costs