Exam 1 Review

15-110 – Monday 09/30

Announcements

- Check3 was due today
- Check2/Hw2 revision deadline **tomorrow (Tuesday) at noon!**
- No Gradescope exercise today (no new material)
- **Exam1 on Wednesday!**
  - Bring something to write with, your andrewID card, and **your andrewID written down on a piece of paper**
  - Arrive **5 minutes early if possible** – we're checking IDs + paper at the door
  - Do not enter until you have been checked in

Announcements – Code Reviews

- **Code reviews!**
  - **What:** meet with a TA for 10-15 minutes to get qualitative feedback on your code from your Hw2 submission. Attending the meeting and actively participating gets you 5 points on Hw3.
  - **Why:** code style and structure are important, but not assessed by the autograder. The TA will point out different ways to solve the problems and areas where you can code more clearly or more robustly
    - Some students may be exempted from this meeting if they already have good style. We'll let you know if you're in that group before sign-ups are released.
  - **When:** this weekend (Saturday-Sunday, a few slots on Monday)
  - **Where:** TA's choice

- How to sign up for a code review slot
  - **Link:** TBA on Piazza
  - **Important:** sign-ups for each TA slot close 5pm Friday
  - **Also important:** don't be late! If you are more than 3 minutes late to your meeting, you will not get credit on Hw3.
    - If something comes up and you need to cancel, notify the TA at least an hour before your timeslot. Do not do this multiple times.

Review Topics

- Nesting control structures
- For Loops & While Loops
- Strings
  - Looping by index
  - Indexing/slicing

Nesting Control Structures

Nesting is the process of **indenting control structures** so that they occur inside other control structures.

So far, we have learned about several control structures: **function definitions**, **conditionals**, **while loops**, and **for loops**.

All of these structures have bodies, and each can be indented so it occurs inside the body of another structure.

Let's talk about common nested structures.

It is common to nest **loops and conditionals inside functions.**

We usually write function definitions at the top level of a program, and nest conditionals/loops inside them when they're needed.

When we return in a nested conditional/loop, we exit that structure and the whole function immediately.

```
def hasVowels(s):
    for i in range(len(s)):
        if s[i] in "aeiou":
            return True
    return False
```

Note how the loop is indented inside the function, and its body is indented again.

If the line '`return True`' is reached, the function will exit immediately without finishing the loop.

It is common to nest **conditionals inside loops.**

We often nest a conditional inside a loop to **check a certain property for every element** that is iterated over.

It's okay to do nothing on iterations that don't meet the requirement if there is no alternative action!

```
def countVowels(s):
    result = 0
    for i in range(len(s)):
        if s[i] in "aeiou":
            result = result + 1
    return result
```

We don't need to update result if the letter isn't a vowel, so do nothing instead.

It is common call **functions inside functions**.

Activity:

What does this print?

```
def foo(a, b):
    y = a + b
    print("y in foo:", y)
    return y + 3

def bar(x):
    y = x + 1
    print("y in bar:", y)
    return foo(x, y)

print(bar(4))
```

It is common to **nest loops inside other loops**.

If you need to **iterate over multiple dimensions**, a nested loop (one loop nested inside another) will manage the complex iteration. Each loop control variable manages one dimension.

```
def coordinates(x, y):
    for xNum in range(x):
        for yNum in range(y):
            print("(" + str(xNum) + ", " +
                        str(yNum) + ")")
```

It is important that the two loop control variables have different names, so that they can be referred to separately!

Loops

A **`while` loop** is a type of loop that keeps repeating only while a certain condition is `True`.

```
while <booleanExpression>:
    <loopBody>
```

indentation (tab) →

```
1  i = 1
2  while i <= 5:
3      print(i)
4      i = i + 1
```

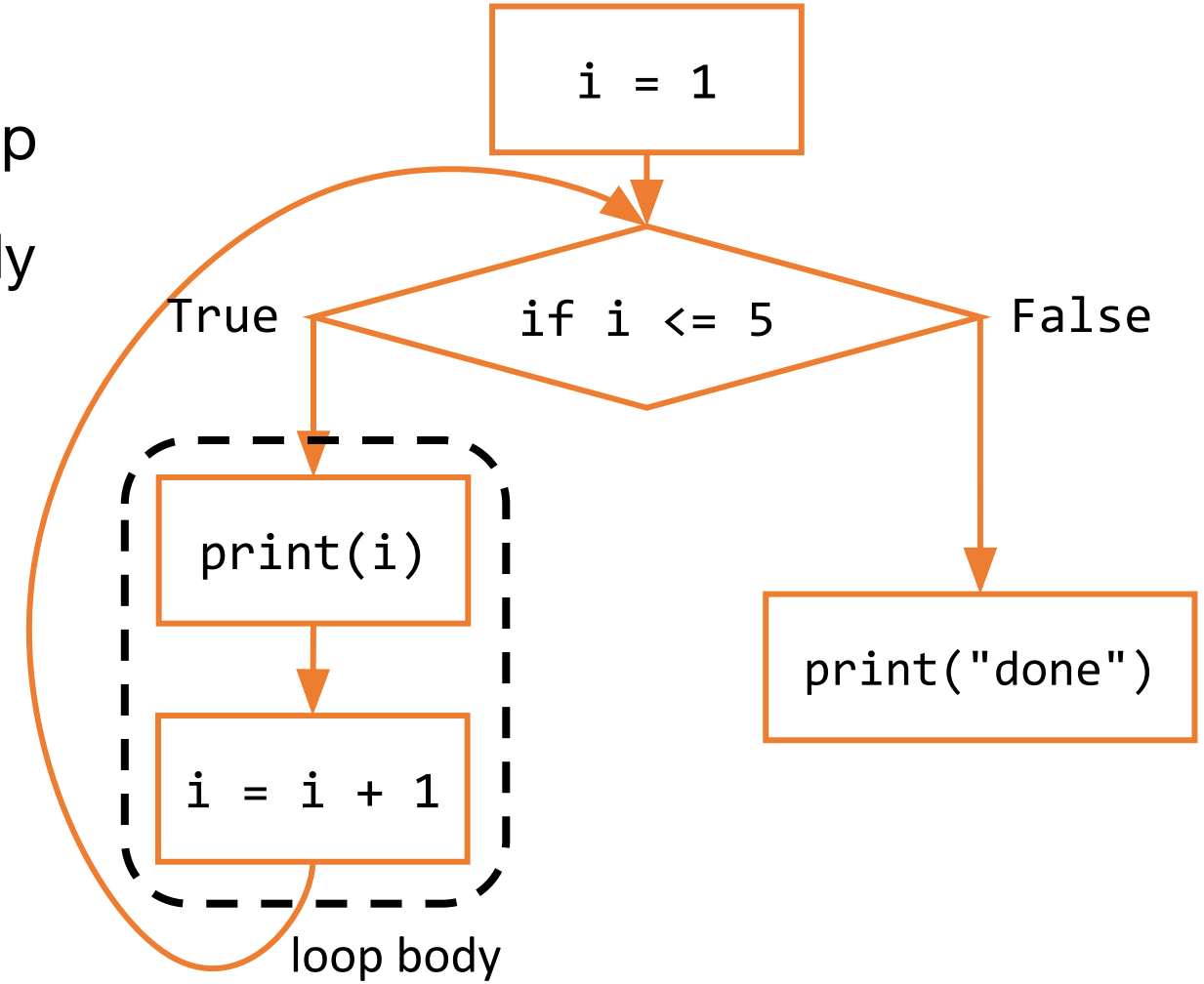
`while` is how Python knows this is a while loop

`:` and indentation is start of **while loop body**

`while` Loop Flow Chart

Unlike an `if` statement, a `while` loop flow chart needs to include a transition from the `while` loop's body back to itself.

```
i = 1
while i <= 5:
    print(i)
    i = i + 1
print("done")
```



To design algorithms with loops, we need to identify what needs to change each iteration by creating a **loop control variable**.

A loop control variable **needs three things to work correctly**: start value, continuing condition, update action

Algorithm: Print numbers 1 to 10 (inclusive)

control variable: `num`

start value: `1`

continuing condition: `num <= 10`

update action: `num = num + 1`

```
num = 1
while num <= 10:
    print(num)
    num = num + 1
```

While Loops – Code Reading

Activity:
What does this code print?

```
n = 3
while n > 0:
    if n == 5:
        n = -100
    print(n)
    n = n + 1
```

A.
```
3
4
```

B.
```
3
4
5
```

C.
```
3
4
-100
```

D.
```
3
4
5
-100
```

While Loops – Code Writing

Activity: Write a while loop where:

Loop control variable is x

The start value is 99

Continuing condition is while x is a positive number

The **update action** is to subtract 2 from x

A **`for` loop** is a type of loop that keeps repeating over a **specific range of values**.

```
for <loopVariable> in range(<maxNumPlusOne>):
    <loopBody>
```

indentation (tab) →

```
1 for i in range(5):
2     print(i+1)
```

`for` is how Python knows this is a for loop and

`i` is the loop control variable

`range(5)` describes the range of values it will loop over: 0, 1, 2, 3, 4

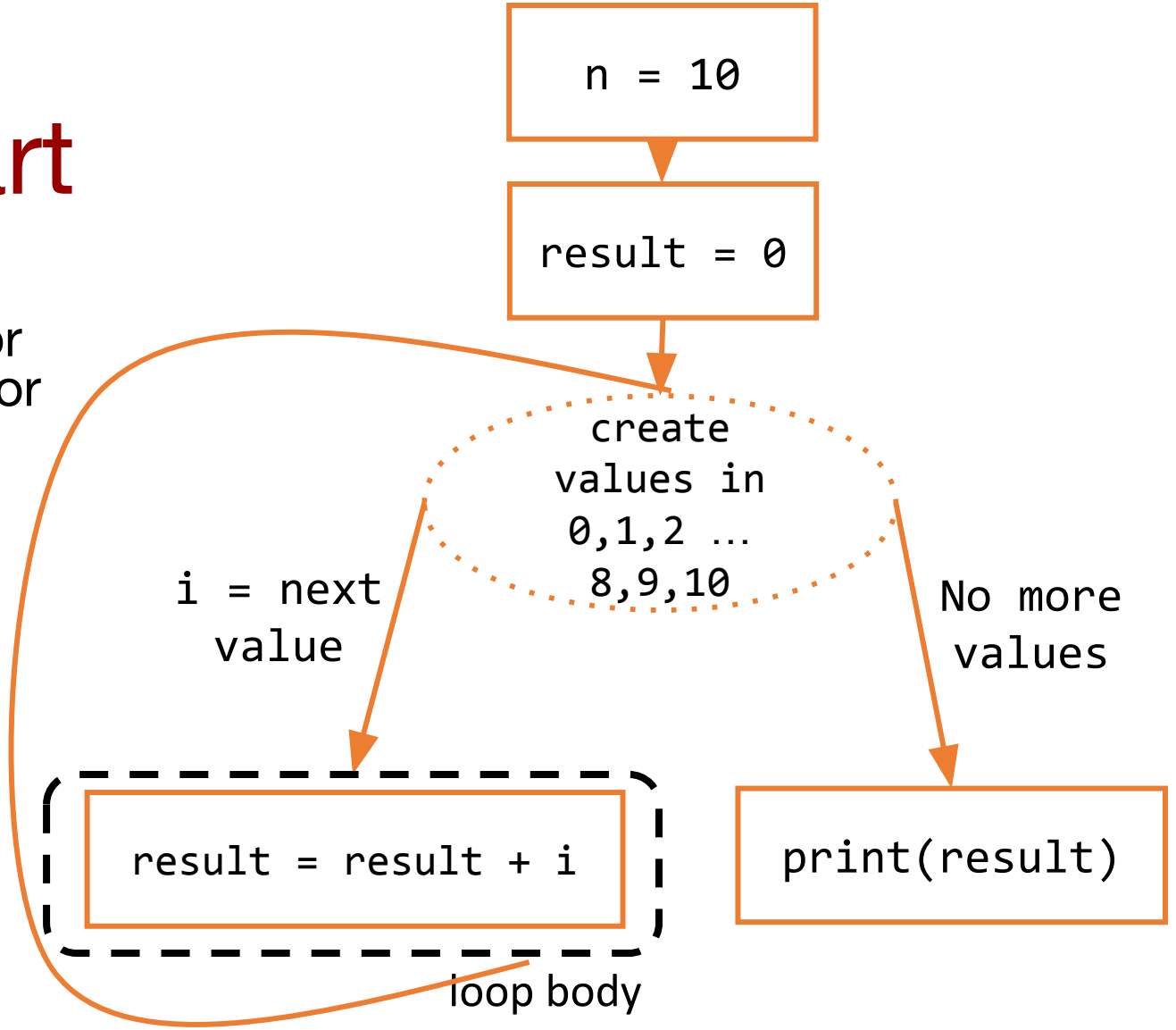
`:` and indentation is start of **for loop body**

For Loop Flow Chart

Unlike while loops, we don't initialize or update the loop control variable. The for loop does those actions automatically.

We show actions done by the range function with a dotted outline here, because they're **implicit**, not written directly.

```
n = 10
result = 0
for i in range(n + 1):
    result = result + i
print(result)
```



`range` has 3 arguments, **two are optional** with default values

```
for i in range(<start>,<stop>,<step>)
```

`<start>` is first value of `i`

optional, default = 0

`<stop>` is last value of `i+1`

required

`<step>` how much to increment `i`

optional, default = 1

```
for i in range(3, 8, 2):
    print(i)
```

	i
iteration 1	3
iteration 2	5
iteration 3	7

For Loops – Code Reading

Activity: Which of the following will add up the numbers from 1 to 4 (inclusive)?

A.
```
for i in range(1,4):
    total = 0
    total = total + i
```

B.
```
total = 0
for i in range(1,4):
    total = total + i
```

C.
```
total = 0
for i in range(1,5):
    total = total + i
```

D.
```
for i in range(1,5):
    total = 0
    total = total + i
```

For Loops – Code Writing

Activity: A Happy Number is a number that is evenly divisible by 2 and is not evenly divisible by 3.

Write a function `totalHappyNumbers(n)` that takes a positive integer `n` and adds up all the happy numbers from 1 to `n` (inclusive) and returns the sum.

Example:
`totalHappyNumbers(6)` returns 6 because 2+4 = 6

`totalHappyNumbers(10)` returns 24 because 2+4+8+10 = 24

Bonus: How can you write `totalHappyNumbers(n)` with a while loop?

Strings

Text values are called **strings**.

Text is recognized by Python as a string by putting it into either **single quotes**: `'Hello'` or **double quotes**: `"Hello"`

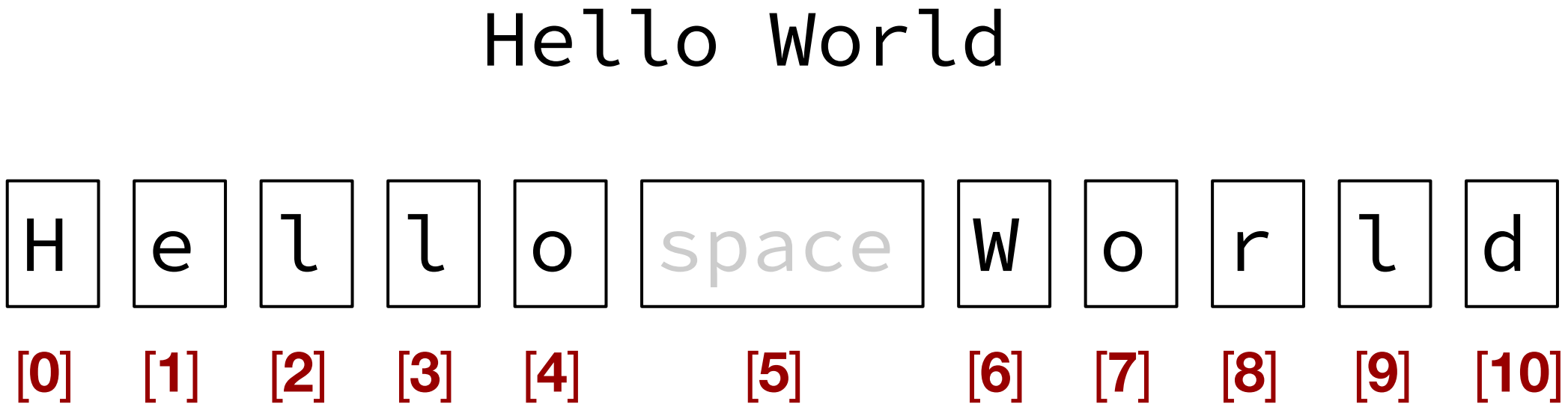
Strings can be **concatenated** using addition operator `+`:

```
>> "Hello" + "World"
HelloWorld
```

Strings can be **repeated** using multiplication operator `*`:

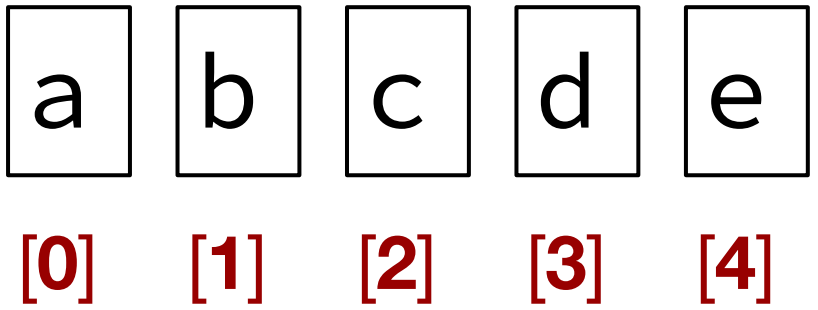
```
>> "Hello" * 3
HelloHelloHello
```

Each character in a string is stored at a specific location **(index)**.



Indices **start at 0**!

We can get a subset of the characters of a string using **slicing**.



Slices are similar to ranges (stop is not inclusive, start and step are optional) but the syntax is **inside square brackets and separated by colons** **[**<*start*>**:**<*stop*>**:**<*step*>**]**

```
s = "abcde"
print(s[2:len(s):1])    # prints "cde"
print(s[0:len(s)-1:1]) # prints "abcd"
print(s[0:len(s):2])    # prints "ace"
```

Slices have 3 parts, **ALL are optional** with default values.

When we want to use a default value for a part, **we leave it blank in the slice**.

`s[::]` and `s[:]` are both the string itself, unchanged

`s[1:]` is the string without the first character (start is `1`)

`s[:len(s)-1]` is the string without the last character (end is `len(s)-1`)

`s[::3]` is every third character of the string (step is `3`)

We can **loop over characters in a string** by visiting each index.

If the string is `s`, the string's first index is `0` and the last index is `len(s)-1`. Use `range(len(s))` to visit all possible indexes.

```
s = "Hello World"
for i in range(len(s)):
    print(i, s[i])
```

Strings – Code Reading

Activity: What are the outputs of the following slices:

```
s = "Towers of Hanoi"


s[3:8]
s[len(s)::-1]
s[10:1]
s[::]
s[len(s)]
```

Strings – Code Writing

Activity: Write `findMatches(s1, s2)`, which takes two same-length strings and returns `True` if they ever have the same character at the same index, and `False` otherwise.

Examples:

`findMatches("apple", "guava")` returns `False`.

`findMatches("apple", "grape")` returns `True`, because the es match.

Exam Taking Tips

- You do not need to answer questions in order!
  - Read all the questions and then decide which ones to answer
  - Be mindful of not spending too much time on one question
- Never leave anything blank
  - If we ask you to write a function, at least write the function definition
- Don’t track stuff in your head, write it down
  - What is the value of each variable after executing each line?
  - What are the indices of each of the elements in a string?
- Pay close attention to restrictions in the writeup
- **Get rest!!!**