KIT100 PROGRAMMING PREPARATION

Lecture Five:

Repetitive Tasks

Lecture Objectives

- Comparing Strings
- Processing Data Using the *for* Statement
  - More about the print function (- *format string %*)
  - *Break / continue* statements
- Processing Data Using the *while* Statement
- Nesting Loop Statements
  - Multiple *for / while* Statements

Comparing Strings

- You can use (> , < , >= , <= , == , !=) to compare two strings. Python compares strings lexico-graphically i.e using the ASCII value of the characters.
- String comparisons are case sensitive
- Suppose you have str1 as "Jane" and str2 as "Jake" . The first two characters from str1 and str2 (J and J) are compared. As they are equal, the second two characters are compared. Because they are also equal, the third two characters (n and k) are compared. And because 'n' has greater ASCII value than 'k' , str1 is greater than str2 .

Takeaway:
String comparison does a compare for each individual letter, working left to right. As characters are actually represented by some underlying numerical encoding e.g. ASCII (or unicode), when we compare that we are actually looking at the numerical value – so that's how we can say for example, 'b' (ASCII 98) is bigger than 'a' (ASCII 97)

Comparing Strings

- More examples:

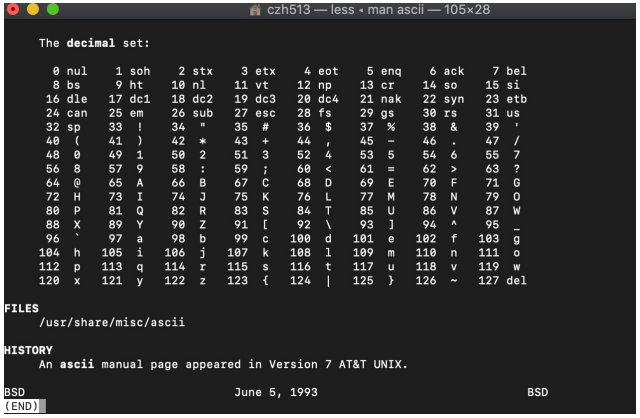
```
>>> "tim" == "tie"
False
>>> "free" != "freedom"
True
>>> "arrow" > "aron"
True
>>> "green" >= "glow"
True
>>> "green" < "glow"
False
>>> "green" <= "glow"
False
>>> "ab" <= "abc"
True
>>>
```

ASCII table

(ASCII-valued) integer Character

0	nul	1	soh	2	stx	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc2	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	sub	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	**0**	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	**65**	**A**	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	**97**	**a**	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	12			

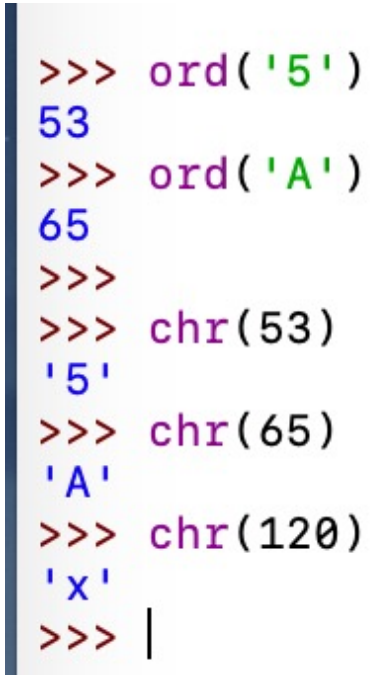
Show the table in the terminal (Mac)
>>> man ascii



ASCII value

- The **`ord()`** function
  - converts a **character** to **its integer (ASCII) value**
- The **`chr()`** function
  - converts an **(ASCII-valued) integer** to a **character**

```
>>> ord('5')
53
>>> ord('A')
65
>>> chr(65)
'A'
>>> chr(120)
'x'
```



Repetitive Tasks in Python

- So far we have worked through examples that have performed a series of steps, once, and then stopped.
- The real world doesn't work this way.
- Many of the tasks that humans perform a repetitious.

Loops

- Most programming languages call any sort of repeating sequence of events a loop.
- Picture the repetition as a circle, with the code going round and round executing tasks until the loop ends.
- Loops are an **essential** part of applications.

Processing data using the for statement

- The first looping code block that most developers use
  - It is the *for* statement.
- A ***for*** loop executes a **fixed number of times**, and **you know the number of times** it will execute before the loop even begins.
- This is the easiest type of loop as everything about a for loop is known at the outset.

Python for statement

- In Python, `for` statements (or loops) are very simple.
- The idea of a `for` loop is rather simple, we just loop through certain code for *a certain number of times*.
- The number of times we loop is pre-determined – we must have a *sequence* of items, and each *iteration* of the loop *consumes* an item/value.
- When no items are left, the loop stops.

Understanding the for statement

- A for loop begins with the `for` statement.
- The `for` statement describes how to perform the loop.

```
for letter in “apple”:
```

colon

variable

sequence

What are the items/values in the sequence?

```
>>> for letter in "apple":
        print (letter)


a
p
p
l
e
>>>
```

Syntax of the for loop

End with a colon/prompt

- The syntax of the `for` loop is:

```
for variable in sequence:
    statements
```

indent

- *variable* is only defined in the block of statements;
- it takes on **one value** (in turn) per iteration, from the given sequence
- *sequence* is 1 or more items
  - Elements from a **list** (defined later in semester)
  - Comma-separated items (e.g. 1,2,3)

```
>>> for letter in "apple":
        print (letter)

a
p
p
l
e
>>>
```

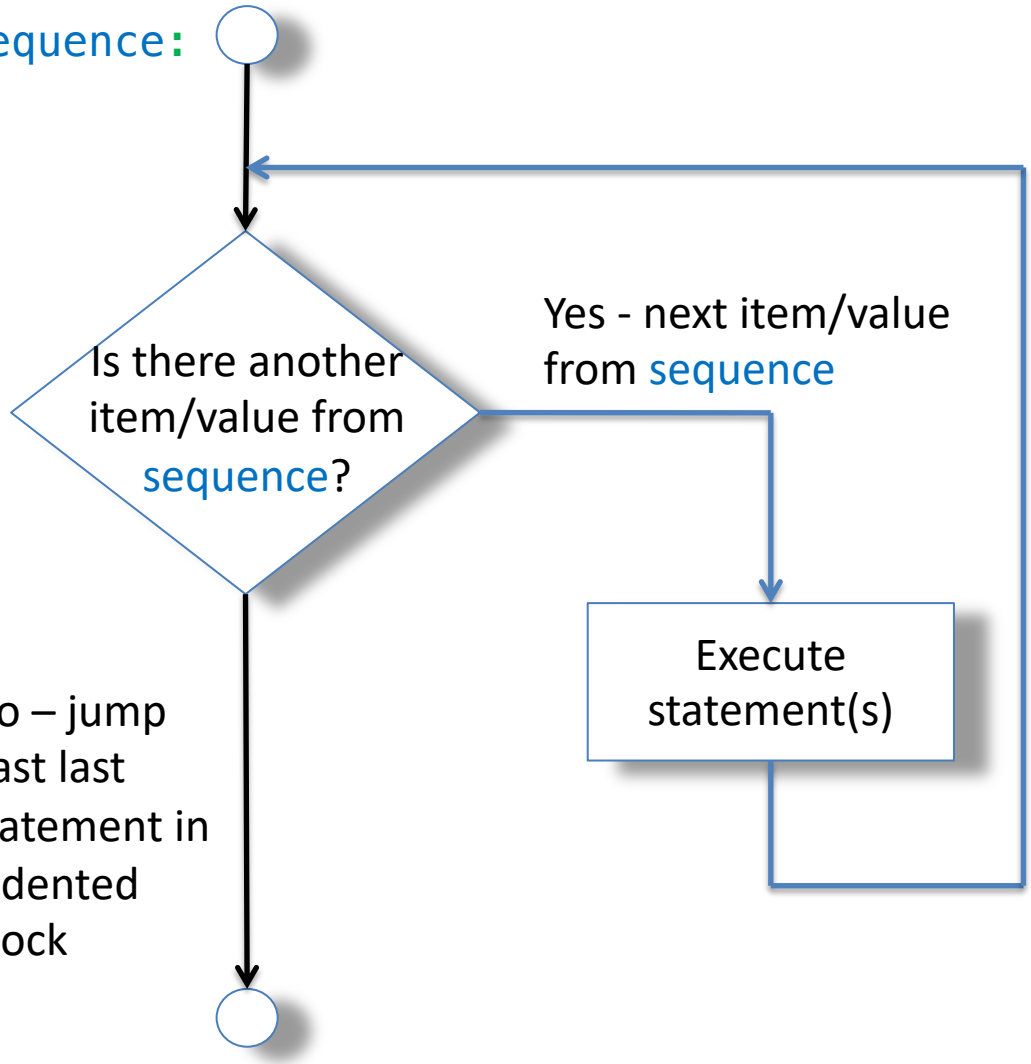
test.py - /Users/

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

```
========== RESTART: /Users/czh
apple
banana
cherry
>>>
```

for loop flowchart

```
for variable in sequence:
```



Creating a basic *for* loop

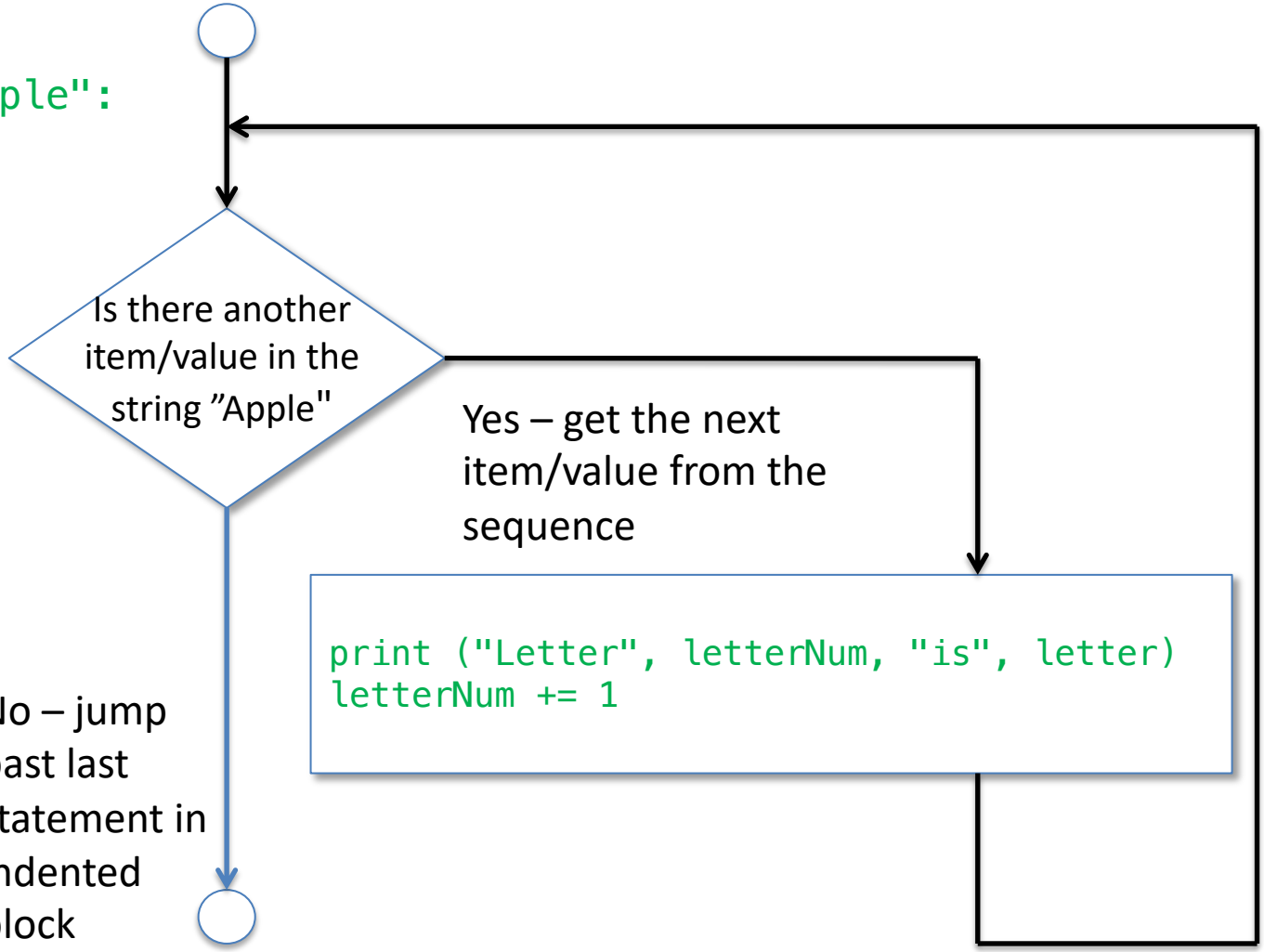
- Let's go back to our `"Apple"` example.
- **Task**
  - We want to write an application that displays the *letter number* and the *current letter* of the string `"Apple"`.
  - We know we have a controllable loop – we can set the count condition to the string `"Apple"`.

Creating a basic for loop

- **Pseudo Code**
  - Create a **variable** called `letterNum`
  - Create a **sequence** in the form of the string `"Apple"`
  - Place the first letter of the string into the variable `letter`
  - Display the ***letter number*** and the ***value*** of `letter` to the user
  - Increment the value of `letterNum` by 1
  - Continue to display the ***letter number*** and the ***new value*** of `letter` to the user until the end of the sequence has been reached

Flowchart for Bazinga for loop


```
letterNum = 1
for letter in ”Apple":
```



Using for loops

```
'''
Title: for loop Example

Purpose: To demonstrate how a for loop works with
         the string 'apple'
'''

letterNum = 1

for letter in "Apple":

  print ("Letter", letterNum, "is", letter)
  # print ("Letter %d is %s" % (letterNum,  letter)) # alternatively - use perce

  letterNum += 1
```

Output

```
Letter 1 is A
Letter 2 is p
Letter 3 is p
Letter 4 is l
Letter 5 is e
>>>
```

The range function

- The `range` function can generate a sequence for us.
  - It's very useful to use with the `for` statement
- `range(x)` will produce a sequence of integers, starting at 0 (zero) up to **one less** than `x`
  e.g. `range(10)` gives `0,1,2,3,4,5,6,7,8,9`
- `range(x,y)` will produce a sequence of integers, starting at `x`, and ending **one less** than `y`
  e.g. `range(1,6)` gives `1,2,3,4,5`
- `range(x,y,z)` will produce a sequence of integers, starting at `x`, ending at **one less** than `y`, and stepping `z` each time
  e.g. `range(1,9,2)` gives `1,3,5,7`

```
>>> for number in range(1,6):
        print(number)


1
2
3
4
5
>>>
```

```
>>> for number in range(1,9,2):
        print(number)


1
3
5
7
>>>
```

```
>>> range(10)
range(0, 10)
>>> for number in range(10):
        print(number)


0
1
2
3
4
5
6
7
8
9
>>>
```

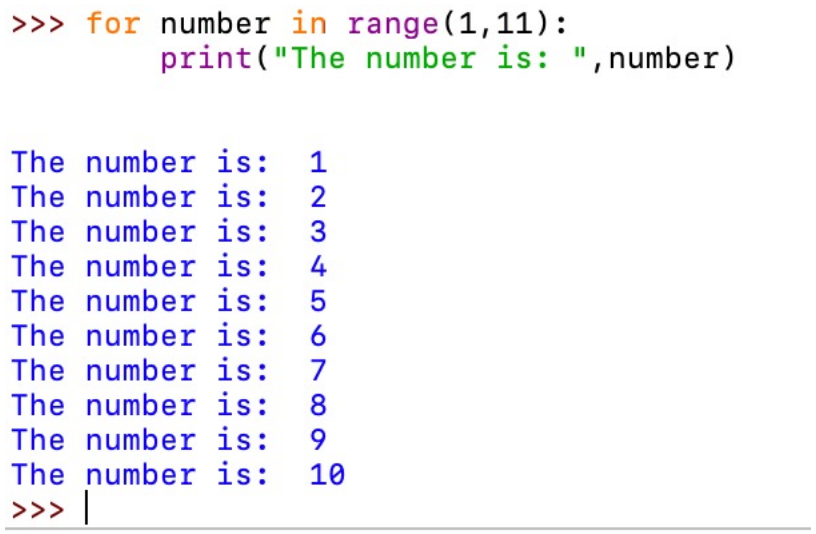
```
>>> for number in 0,1,2,3,4,5,6,7,8,9:
        print(number)


0
1
2
3
4
5
6
7
8
9
>>>
```

range and for example

```
for number in range(1,11):
    print("The number is: ",number)
```

Output:

```
The number is: 1
The number is: 2
The number is: 3
The number is: 4
The number is: 5
The number is: 6
The number is: 7
The number is: 8
The number is: 9
The number is: 10
```



More about the print function

- As we've seen, the Python 3 `print` function will allow you to combine different *types* of output if you **separate** them by a comma.

  e.g.

```
a = 1            # <int>
b = 2.7          # <float>
c = "hello"      # <str>
print (c,a,b)
```

```
>>> a = 1
>>> b = 2.7
>>> c = "hello"
>>> print (c, a, b)
hello 1 2.7
>>>
```

- The output will show the values, but each value will be separated by a space

  i.e.

```
hello 1 2.7
```

Space

More about the print function

- Sometimes you may **NOT** want this introduced space. **One** way to do that as we have seen before is to convert everything you want to print into a string (using the `str` function), and then use the `+` operator (this is *string concatenate/join*, not addition!) to include them:

e.g.

```
a = 1                  # <int>
b = 2.7                # <float>
c = "hello"            # <str>
print (c + str(a) + str(b))
```

```
>>> a = 1
>>> b = 2.7
>>> c = "hello"
>>> print (c + str(a) + str(b))
hello12.7
>>>
```

output:

```
hello12.7
```

More about the print function

- You can also **stop** `print` creating a **new line** after the output:

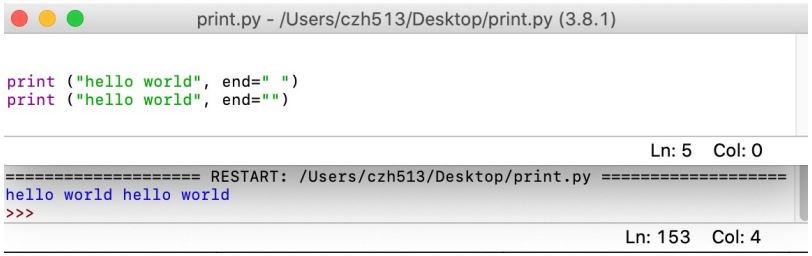
e.g.

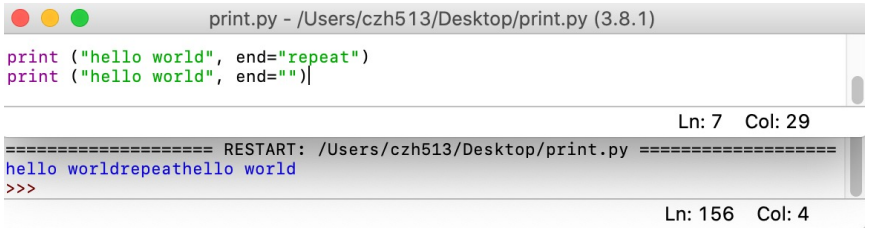
```
print ("hello world", end=" ")
print ("hello world", end="")
```

"what character should be added to the end of the line?"

output:

```
hello world hello world
```





- By default (if you don't specify it), `end = "\n"` (which is the ***newline*** character)

Controlling print output – *format string*

- Another way we can control the output of the print function, and "embed" variables inside the double quotes ("), is to use a ***format string***.
- A special property of strings is they have a built-in operator
  - the ***format string*** % operator

    (don't confuse with the modulus operator, also %)
- Values can be included inside the ***format string*** by using specialised formatting sequences as **placeholders**.
  - The format operator % matches and replaces **each** format sequence with a value you provide afterwards in a bracketed, comma separated list (x, y, z,… etc)

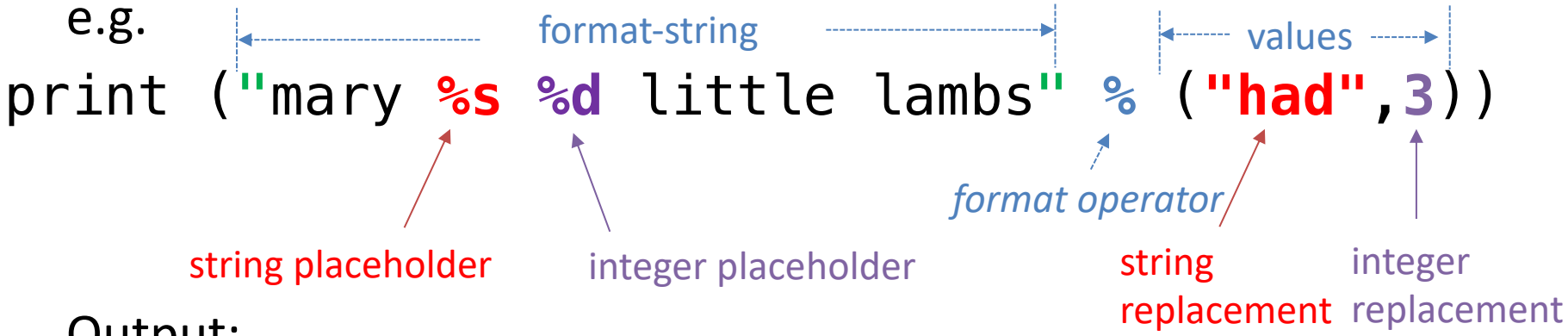
%d	Represents an integer as a string
%s	Represents a string value as a string
%f	Represents a float as a string
%c	Represents a single character as a string

Controlling `print` output - *format*

- The form is ***`format-string % (comma-separated values)`***

  where ***`format-string`*** is the string to print; ***`values`*** is a list of values to be placed inside the format-string when the special format sequences are found, `%` is the format operator:

  e.g.



  Output:

```
mary had 3 little lambs
```

```
>>> print ("mary", str, num, "litte lambs")
mary had 3 litte lambs
>>> print ("mary " + "had " + "3 " + "little lambs")
mary had 3 little lambs
>>> print ("mary %s %d little lambs" % ("had", 3))
mary had 3 little lambs
>>> print ("mary %s %s %d little lambs" % ("had", "more than", 3))
mary had more than 3 little lambs
>>>
```

Controlling print output - *format string*

- The formatting sequence
  - specify the ***field width*** (the output value is padded with spaces) and justification (**right justification** is the default, use '-' for left justification) before the specific type's character

e.g.

%*width*d an integer, printed with *width* characters

```
print ("Right justified number:%6d shown" % 32)
Output: Right justified number:     32 shown
print ("Left justified number: %-6d shown" % 32)
Output: Left justified number:32     shown
```

6 chars wide

```
>>> print ("normal number:%d shown" % 32)
normal number:32 shown
>>> print ("Right justified number:%6d shown" % 32)
Right justified number:     32 shown
>>> print ("Left justified number: %-6d shown" % 32)
Left justified number: 32     shown
>>>
```

Back to using for loops (now using a format string)

"Apple" example using the format string

```
'''
Title: for loop Example

Purpose: To demonstrate how a for loop works with
         the string 'apple'
'''

letterNum = 1

for letter in "Apple":

  # print ("Letter", letterNum, "is", letter)
  # alternatively - use operator control print %
  print ("Letter %d is %s" % (letterNum,  letter))

  letterNum += 1
```

Output

Python 3.8.1 Shell

```
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week5/apple.p
y
Letter 1 is A
Letter 2 is p
Letter 3 is p
Letter 4 is l
Letter 5 is e
>>>
```

The break statement

- The `break` clause makes **breaking out of** a loop possible.
- The `break` clause ~~needs to~~ **must** be surrounded with an `if` statement that defines the condition for issuing a break.
- We can expand on our `for` loop example with a `break` clause and getting some user input.
- Use `break` **very sparingly** – usually you can write code that **doesn't** require it.

apple_break.py - /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/we...

```
'''
Title: for break Example

Purpose: To demonstrate how a for loop works with
         the string 'apple'
'''

for letter in "Apple":
  print (letter)
  break
```

Ln: 10 Col: 7

```
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week5/apple_b
reak.py
A
>>>
```

apple_break.py - /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/we...

```
'''
Title: for break Example

Purpose: To demonstrate how a for loop works with
         the string 'apple'
'''

for letter in "Apple":
  print (letter)
  if (letter == 'l'):
    break
```

Ln: 11 Col: 9

```
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week5/apple_b
reak.py
A
p
p
l
>>>
```

Example break clause


```
'''
Title: break in for loop Example
Purpose: To demonstrate how a break clause works
         in a for loop
'''

value = input("Type less than 7 characters:")
letterNum = 1

for letter in value:
    print ("Letter %d is %s" % (letterNum, letter))

    letterNum += 1
    if letterNum >= 7:

        break
        print ("The string you entered is too long!")

# print("This is outside the for loop")
```

Ln: 2 Col: 22

This code has a deliberate semantic error… can you see where it is?

break clause output

Output 1

```
Type less than 7 characters:abcd
Letter 1 is a
Letter 2 is b
Letter 3 is c
Letter 4 is d
>>>
```

Output 2

```
Type less than 7 characters:abcdefghijk
Letter 1 is a
Letter 2 is b
Letter 3 is c
Letter 4 is d
Letter 5 is e
Letter 6 is f
>>>
```

There's a problem with our output in the second example above – can you see what it is? Where's our `"The string you entered is too long"` message?

The continue statement

- Sometimes you want to check every element in a sequence but don't want to process certain elements (you want to skip them)
- As with the `break` clause, the `continue` clause **ALWAYS** appears as part of an `if` statement.
- Use `continue` **very sparingly** (if ever!). Usually you can write your code in such a way that `continue` is not needed.

continue clause example

```
'''
Title: continue in for loop Example
Purpose: To demonstrate how a continue clause
         works in a for loop
'''
letterNum = 1

for letter in "apple_orange":
    if letter == "n":
        print ("Encountered n, not processed.")
        continue
    print ("Letter %2d is %s" % (letterNum, letter))

    letterNum += 1
```

Ln: 10 Col: 19

Output

```
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week5/continu
e.py
Letter  1 is a
Letter  2 is p
Letter  3 is p
Letter  4 is l
Letter  5 is e
Letter  6 is _
Letter  7 is o
Letter  8 is r
Letter  9 is a
Encountered n, not processed.
Letter 10 is g
Letter 11 is e
>>>
```

Ln: 62 Col: 4

continue clause output

if print statement after continue instead of before..

continue.py - /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week5...

```
'''
Title: continue in for loop Example
Purpose: To demonstrate how a continue clause
         works in a for loop
'''
letterNum = 1

for letter in "apple_orange":

    if letter == "n":
        continue
        print ("Encountered n, not processed.")

    print ("Letter %2d is %s" % (letterNum, letter))

    letterNum += 1
```

Ln: 11 Col: 16

Output

```
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week5/continu
e.py
Letter  1 is a
Letter  2 is p
Letter  3 is p
Letter  4 is l
Letter  5 is e
Letter  6 is _
Letter  7 is o
Letter  8 is r
Letter  9 is a
Letter 10 is g
Letter 11 is e
>>>
```

Ln: 46 Col: 14

Default!

The else clause on loops

- The **`else`** clause can be used with **loops** as a final series of statements to execute **when the loop has finished iterating** (it's also known as a *completion clause*).
- **The only time the `else` clause isn't executed** is - if you used `break` to exit the loop.
- A useful example
  - If you are using a loop to search a sequence for something, if you find what you are looking for, you exit the loop with a break statement - **the `else` clause is not executed**.
  - If you don't find what you are looking for, eventually when the loop finishes, you execute the `else` clause – here you might take some action to indicate the search was unsuccessful (the example attached)
- It is very rare to write code needing the `else` clause (you will **not** need it!)

In this unit, we don't use this else clause in loops in our portfolio tasks

else clause Example

```
print ("I search for the 'a' character.")

value = input("Type less than 7 characters: ")

letterNum = 1

for letter in value:

    print ("Letter %d is %s" % (letterNum, letter))

    letterNum += 1

    if letter == "a":
        print ("You typed an 'a'!")
        break

else:
    print ("You did not type any 'a' characters.")
```

Only executed if we didn't 'break' out of the for loop

Remember – we're not using this syntax in the unit…

else clause output

Output1

```
I search for the 'a' character.
Type less than 7 characters: abcd
Letter 1 is a
You typed an 'a'!
>>>
```

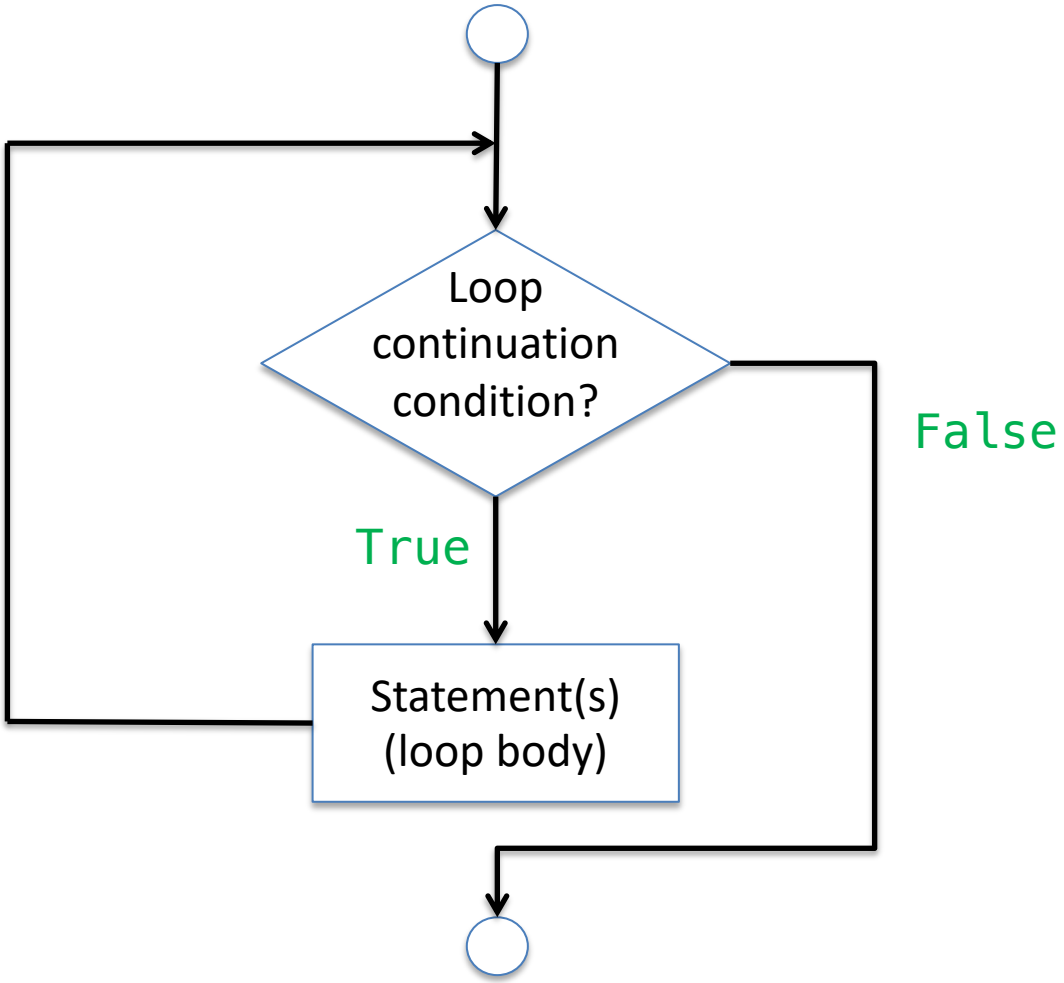
Output2

```
I search for the 'a' character.
Type less than 7 characters: cdef
Letter 1 is c
Letter 2 is d
Letter 3 is e
Letter 4 is f
You did not type any 'a' characters.
>>>
>>>
>>>
>>>
```

Understanding the while statement

- A **`while`** loop executes statements repeatedly as long as a **condition remains true**.
- The "**condition**" is only **checked once at the start of each iteration (loop)** (it is not checked continuously)
- The while statement works with a **condition** rather than a sequence.
- The **condition** states that the **`while`** statement should perform a task until the condition is **no longer true**.

while loop flowchart



Alternatives to using the while statement

- Task: consider for example, we want to get Python to display the string `"Hello World!"` 5 times.

- One way to achieve this is to simply type the `print` function 5 times (now imagine doing this 50,000 times?)

```
print ("Hello World!")
print ("Hello World!")
print ("Hello World!")
print ("Hello World!")
print ("Hello World!")
```

- Another way is to use the **string** multiply operator!

```
print ("Hello World!\n"*5)
```

Repeat the string 5 times

Using the while statement

- We can make our programs more robust by removing repetitious lines of code by using the `while` statement.

- Group the code you want repeated as an indented block of statements underneath the `while` keyword
  - you then need to consider what condition(s) is/are appropriate to keep the loop running (it should eventually stop!)

Syntax for while statement

- The syntax for the *while* statement is:

End with a colon/prompt

```
while loop-continuation-conditions:
    loop-body-statement(s)
```

indent

Using while loops

- We can write our new solution as follows:

  Problem: Display `"Hello World"` as an output 5 times

Pseudo code:

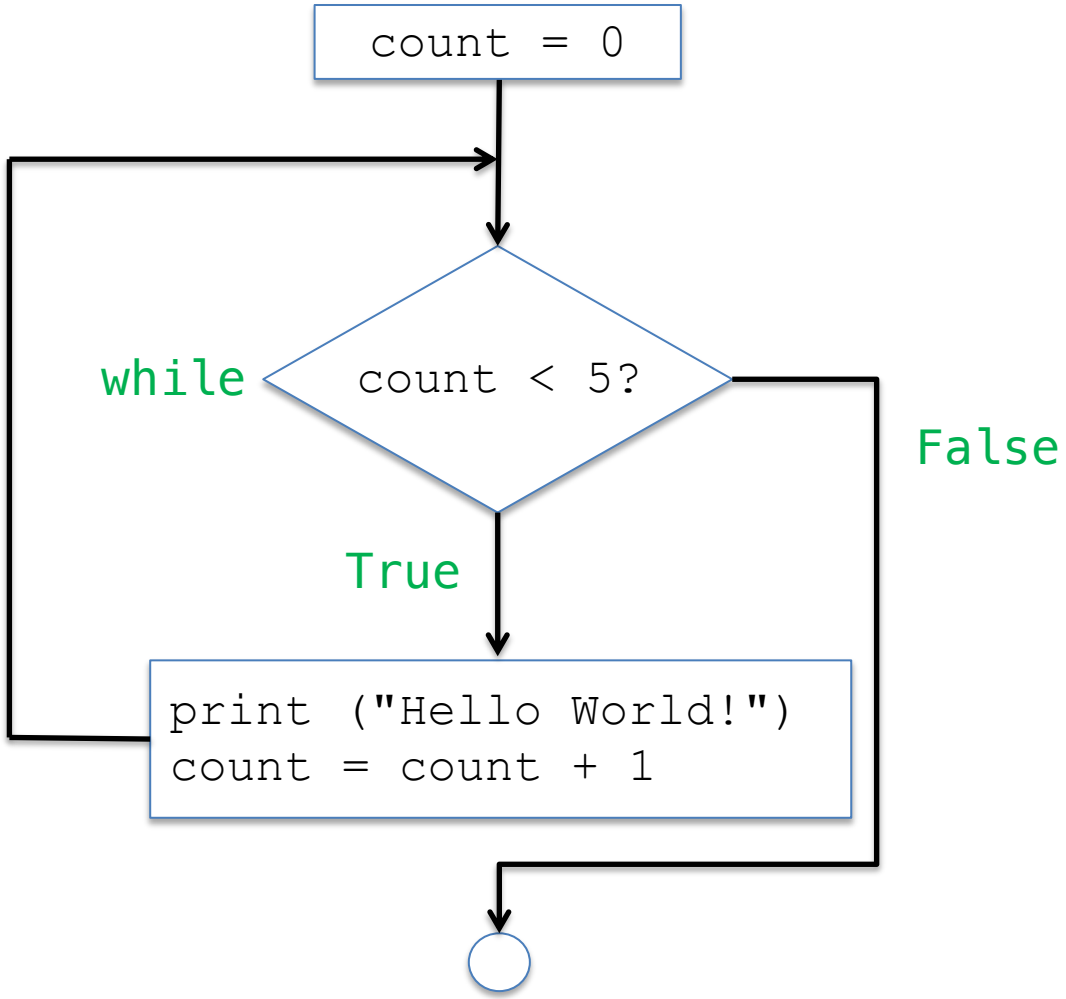
Create a count variable called `count`

Display `"Hello World!"` to the user

Increment the value of `count` by 1.

Keep displaying `"Hello World"` to the user until the value of `count` reaches 5

Flowchart of our problem



Using while loops

```
count = 0
while count < 5:
    print("Hello world")
    count = count + 1
```

Ln: 4 Col: 20

Output

```
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week5/while_p
43.py
Hello world
Hello world
Hello world
Hello world
Hello world
>>>
```

Infinite loop!

```python
'''
Title: Invalid While loop example
Purpose: To use the tutorial two 'Hello World!'
         example in while loop.
'''

count = 0
while count < 5:
    print ("Hello World!")
count = count + 1
```

Can you see why this loop never stops? Trace through the code and see what the value of count is each time we loop..

Our statement of count = count + 1 is not in the loop body.

We need to remember that the entire body must be indented inside the loop.

off-by-one error

- Another common error is
  - to execute a loop **one time** too many or **one time** too few than intended.
- This is known as a ***off-by-one*** *error*.
- This is commonly caused by using the <= or >= operators, rather than the < or > operators
- e.g. if you want your loop to iterate *n* times and you have initially set your *count* variable to 0, then you'll need to use the < operator to get the correct logic. Why? *Assume n is some integer*:

How many times does this iterate?

```
count = 0
while count <= n:
   print("Iteration number:",count)
   count = count + 1
```

```
count = 0
while count < n:
   print("Iteration number:",count)
   count = count + 1
```

Loop design strategies

- Writing a loop that works correctly is not always an easy task.

- There are three steps involved when writing a loop:
  1. Identify the statement that need to be **repeated**
  2. Wrap these statements in a loop

```
while True:
    step1-statements
```

This iterates forever, an infinite loop! because the condition is always true.

  3. Add the "loop continuation condition" and **also add appropriate statements** to the loop body for "changing the loop continuation condition[1] "

```
while loop-continuation-condition:
    step1-statements
    statement(s)-to-change-condition
```

[1] *If you forget to do this, the loop may never stop!*

Controlling the while loop

- You can also control the *`while`* loop with the same three clauses introduced in the *`for`* loop:

  `break`: Ends the *current* **loop**

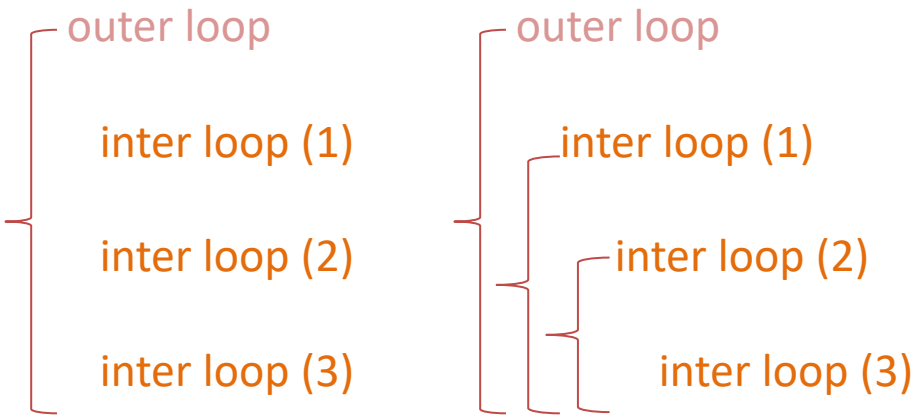
  `continue`: Immediately ends processing of the *current* **element/item**

  `else`: Provides a final series of statements to execute **when the loop body has finished looping**.

*Try to always **avoid** using any of these in your portfolio tasks*

Nesting loop statements

- A **nested loop**
  - a loop that is inside another loop.
  - consist of an *outer* loop, and one or more more *inner* loops.
- Each time the outer loop is repeated, the inner loops are re-entered and started again.



Nested loop Example

- A common example used to show nested loops is a multiplication table.
- We can create our **multiplication table** by nesting a `for` loop inside another `for` loop as we know exactly how many times to iterate (repeat) the body of the loops.
- We can also add some simple formatting…

Multiplication Table

×	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

Nested loop example

- **Task: create a multiplication table using the nested loop**

- **Pseudo Code:**

  Create a control variable `x` and assign it the value 1

  Create a control variable `y` and assign it the value 1

  Display a title line `"Multiplication Table"`

  Use formatting so the title is separated from the table

  Display `x` as the left operand at the beginning of each line
  when the value of `x` is in the range 1 - 10

  Display the product of `x` and `y` `(equal to x multiply y)`

  Increment `y` by 1 using for loop

  Keep looping for as long as the `y` variable is in the range 1 - 10

Nested loop example

nestedloop_p51.py - /Users/czh513/Desktop/KIT0...

```
print ("          Multiplication Table")
print ()
print ("-----------------------------------------------

for x in range(1, 10):
    print ("%3d" % x, "|", end="")

    for y in range (1,10):
        print ("%3d" % (x*y), end="")

    print ()
```

Ln: 13 Col: 12

Output

```
          Multiplication Table

-------------------------------------------------------
  1 |  1  2  3  4  5  6  7  8  9
  2 |  2  4  6  8 10 12 14 16 18
  3 |  3  6  9 12 15 18 21 24 27
  4 |  4  8 12 16 20 24 28 32 36
  5 |  5 10 15 20 25 30 35 40 45
  6 |  6 12 18 24 30 36 42 48 54
  7 |  7 14 21 28 35 42 49 56 63
  8 |  8 16 24 32 40 48 56 64 72
  9 |  9 18 27 36 45 54 63 72 81
>>>
```

Questions?