CS4650 Topic 8:
Intro to Python

Thanks Jonathan

- These slides were contributed by Jonathan Johannsen, formerly of Cal Poly, then at Cypress College, but now at Mt. SAC.

An Example Python Program

- To begin our review of Python, let's write simple a program that thinks of a number and asks the user to guess what it is.
  - First, the user will be asked for an upper and lower bound on the number the computer should think of.
  - Then, a random number between those two bounds will be generated.
  - A `while` loop will continue until the user guesses the number correctly.
    - If they guess too high or low, we will inform them of this.
  - After they guess correctly, we will let them know how many guesses they took.

An Example Python Program

```python
import random

def main():
    """ Asks the user for the bounds of the range of numbers and lets the user guess
    the computer's number until the guess is correct """
    lower = int(input("Enter the lower bound: "))
    upper = int(input("Enter the upper bound: "))

    computer_num = random.randint(lower, upper)
    count = 0

    # Loop until the user guesses correctly
    while True:
        count += 1
        user_guess = int(input("Enter your guess: "))
        if user_guess < computer_num:
            print("Too small!")
        elif user_guess > computer_num:
            print("Too large!")
        else:
            print(f"You got it in {count} tries!")
            break

main()
```

Let's look at this program's statements in more detail.

An Example Python Program

```
import random
```

- An **import** statement tells Python to include the code from another **module** in our program.
  - A module is a file which may contain objects, functions, or classes.
- In this example, we import the `random` module, which includes functions for generating pseudorandom numbers.
- When we use an import with this syntax, we type the module name, dot operator, and function name to access a function from the module.
- For example, to call the `randint` function:

```
my_number = random.randint(lower, upper)
```

An Example Python Program

```
def main():
```

- This line of code is a **function header**.
  - The function is named `main`.
- Although not required, a lot of Python programmers like to place all of their code inside a function like this.
- **Note**: If you follow this convention, don’t forget to add a call to the function in your script at the first level of indentation:

```
main()
```

The input Function

```
lower = int(input("Enter the lower bound: "))
```

- The `input` function pauses program execution and waits for the user to type a response and then the ‘enter’ key.
- The characters the user typed are returned as a `str` object.
- In this example, we pass the returned string to the `int` function.
- This function converts its argument to an integer (if possible).
  - If not possible (for example the user enters “Hello”), an `Error` is raised.
- We often want to **cast** user input from a string to another data type if required by our program.
- Finally, the value returned by `int` is assigned to `lower`.

The input Function

```
my_number = random.randint(lower, upper)
```

- The `randint` function from the `random` module returns an integer between its two arguments (both values are inclusive)
- **Note**: If the first argument is larger than the second, an `Error` is thrown.
  - In a more robust program, we would ensure the user does not input such values.

while Loops

```
while True:
    count += 1
    user_guess = int(input("Enter your guess: "))
    if user_guess < computer_num:
        print("Too small!")
    elif user_guess > computer_num:
        print("Too large!")
    else:
        print(f"You got it in {count} tries!")
        break
```

- A `while True` loop is an example of an **infinite loop**.
  - i.e. Since “true” is always “true”, it will loop until a `break` statement is reached.
- Loops like this are common when we do not want program execution to proceed until the user has entered proper data.
- This particular loop will continue until the user guesses correctly and the final `else` clause is entered.

Formatted String Literals

- Python 3.6 introduced **F-Strings**, or **formatted string literals**, which help us mix string literals and variables in a simpler way.
  - As we will soon see, we can also easily format variables within f-strings.
- The simplest f-string is a string literal with the letter f before it:

```
f'Hello World'
```

- We can pass an f-string to the `print` function, just as we would a normal string literal:

```
print(f'Hello World')    # prints 'Hello World'
```

Placeholders

- While a simple f-string might seem useless, **placeholders** make them more powerful.
- Inside curly braces, we can place a variable:

```
name = "Jonathan"
print(f'Hello {name}!')    # prints 'Hello Jonathan!'
```

- The value the variable holds will be appended to the string that prints.
- As the strings we print become more complicated and contain more variables, f-strings will make the job much easier.
- **Note**: A placeholder *must* have the name of an existing variable, otherwise an error will occur:

```
name = "Jonathan"
print(f'Hello {names}!')    # ERROR!
```

Placeholder Expressions

- What will the following code print?

```
num = 10
print(f'The value is {num * 2}')
```

Placeholder Expressions

- What will the following code print? `The value is 20`

```
num = 10
print(f'The value is {num * 2}')
```

- A placeholder can contain any expression, not just a single variable.
- In this example, we take the value in `num`, multiply it by 2, and attach the result to an f-string, and print it out.

Formatting Placeholder Values

- A **format specifier** can be used to change a placeholder's formatting.
- To do this, we use the following general format:

```
{placeholder: format-specifier}
```

- If the format specifier we place after the colon is recognized by Python, the output will be formatted in a special way. Otherwise, an error will occur.

Formatting Placeholder Values

- What will the following code print?

```
total_cost = 5000
monthly_payment = total_cost / 12
print(f'Your monthly payment is {monthly_payment}')
```

Formatting Placeholder Values

- What will the following code print?

```
Your monthly payment is 416.6666666666667
```

```
total_cost = 5000
monthly_payment = total_cost / 12
print(f'Your monthly payment is {monthly_payment}')
```

- The value that prints looks horrible!
- A precision specifier can be used to tell Python how many digits to include:

```
print(f'Your monthly payment is {monthly_payment: .2f}')
```

  - **.2** tells Python to include two values after the decimal point.
  - **f** tells Python that the number of digits after the decimal point should be fixed (i.e. *always* print two digits, even if they are 0.
  - Note that the value is also properly rounded.

Inserting Comma Separators

- A comma format specifier places commas where needed in large numbers.
- For example:

```
population = 17542142
print(f'The population is {population:,}')
```

Prints:

```
The population is 17,542,142
```

- From right-to-left, a comma is placed after every three digits.
- If the number is three digits or less, no comma will be placed.

Combining Format Specifiers

- In programming, it is quite common to print amounts of money.
- A typical way to print money is to include comma separators (for large amounts of money) and two digits after the decimal point.
- We can do this by combining two format specifiers:

```python
monthy_pay = 5000.0
annual_pay = monthy_pay * 12
print(f"Your annual pay is ${annual_pay:,.2f}")
```

- Note that we have to include the comma format specifier before the precision designator. Otherwise, an error will occur.

Specifying a Minimum Field Width

```
num1 = 127.899
num2 = 3465.148
num3 = 3.776
num4 = 264.861

print(f'{num1:.2f} {num2:.2f}')
print(f'{num3:.2f} {num4:.2f}')
```

- Consider this code. We have four numbers, printing in two columns.
- The problem is, they are not the same length and will print misaligned:

```
127.90 3465.15
3.78 264.86
```

Specifying a Minimum Field Width

- With a minimum field width specifier, we can specify the minimum number of spaces used to display a value.
- To do this, we simply add a number after the placeholder:

```
print(f'The number is {num1: 10}')
```

```
The number is    127.899
```

- Even though the value that prints requires 7 spaces, three extra spaces are added to its left to bring the total to 10.

Specifying a Minimum Field Width

```python
num1 = 127.899
num2 = 3465.148
num3 = 3.776
num4 = 264.861

print(f'{num1:.2f} {num2:.2f}')
print(f'{num3:.2f} {num4:.2f}')
```

- How can we update this code to make it print more neatly?

Specifying a Minimum Field Width

```
num1 = 127.899
num2 = 3465.148
num3 = 3.776
num4 = 264.861

print(f'{num1:10.2f} {num2:10.2f}')
print(f'{num3:10.2f} {num4:10.2f}')
```

- By having each placeholder fill a minimum number of spaces, the columns will now be uniform in size:

```
    127.90    3465.15
      3.78     264.86
```

- The minimum field width specifier must be placed before the precision specifier.
- This only works if the minimum width is larger than the largest number!

Reference Variables and Objects

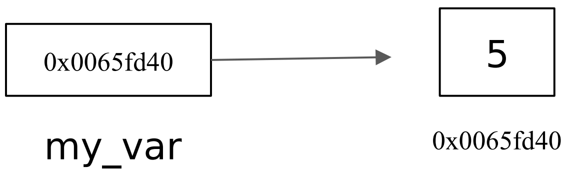
- Everything in Python is an **object** which is stored in the computer's memory.
- We use **reference variables** to **refer** (or **point**) to these objects.
- A Python reference variable can refer to any type of object during its lifetime.
- For example, in the following three statements, the reference variable `my_var` refers to three different objects:

```
my_var = 5                      # my_var points to an int object
my_var = [1, 2, 3]   # my_var now points to a list object
my_var = "Hello"     # my_var now points to a string object
```

Reference Variables and Objects

```
my_var = 5                  # my_var points to an int object
my_var = [1, 2, 3]   # my_var now points to a list object
my_var = "Hello"     # my_var now points to a string object
```

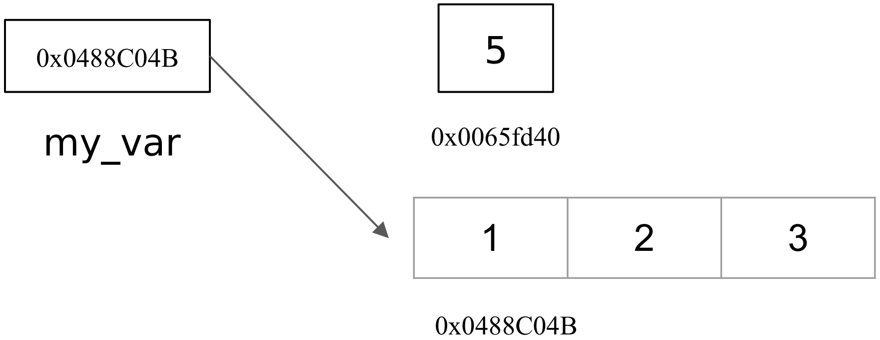
- A reference variable contains the memory address of an object.
- In the following (rather simplified) diagram, we see that the variable `my_var` contains the memory address of an integer object. That object holds the value 5.



Reference Variables and Objects

```
my_var = 5                      # my_var points to an int object
my_var = [1, 2, 3]  # my_var now points to a list object
my_var = "Hello"    # my_var now points to a string object
```

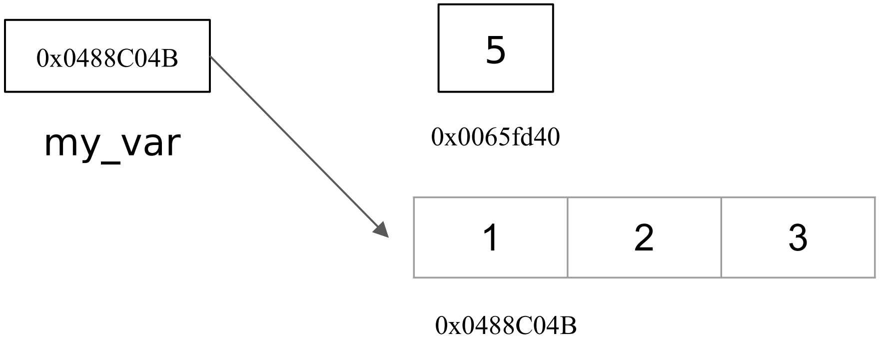
- The right side of an assignment expression is always evaluated first.
- In the second statement, a `List` object is created by the expression to the right of the = operator and placed somewhere in memory (0x0488C04B in this example).
- Its memory address is then assigned to `my_var`.



Reference Variables and Objects

```
my_var = 5                # my_var points to an int object
my_var = [1, 2, 3]   # my_var now points to a list object
my_var = "Hello"     # my_var now points to a string object
```

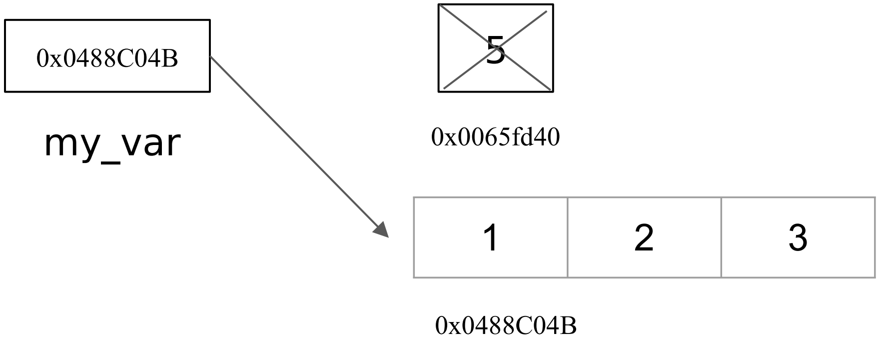
- What happens to the integer object at address 0x0065fd40 at this point?



Garbage Collection

```
my_var = 5               # my_var points to an int object
my_var = [1, 2, 3]   # my_var now points to a list object
my_var = "Hello"     # my_var now points to a string object
```

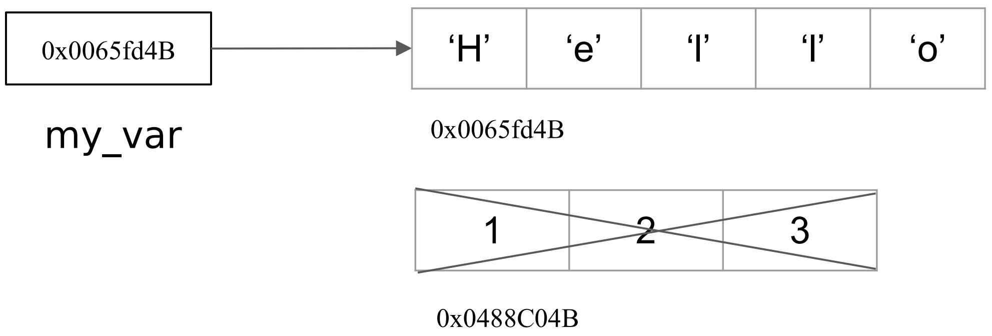
- What happens to the integer object at address 0x0065fd40 at this point?
- It is **garbage collected** (i.e. deleted from memory)
- Once an object has no variable referencing it, Python deletes it from memory, freeing that memory for future objects.



Garbage Collection

```
my_var = 5                  # my_var points to an int object
my_var = [1, 2, 3]   # my_var now points to a list object
my_var = "Hello"     # my_var now points to a string object
```

- In the third statement, a string object is created. Its memory address is assigned to `my_var`.
- The `List` object at address 0x0488C04B is garbage collected.



Garbage Collection

```
my_var = 5
my_var2 = my_var
my_var = 10
print(my_var2)
```

- What will this code print?

- Are any objects garbage collected?

Garbage Collection

```
my_var = 5
my_var2 = my_var
my_var = 10
print(my_var2)
```

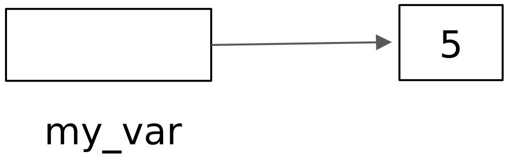
- What will this code print? **5**

- Are any objects garbage collected? **No**!

Garbage Collection

```
my_var = 5
my_var2 = my_var
my_var = 10
print(my_var2)
```

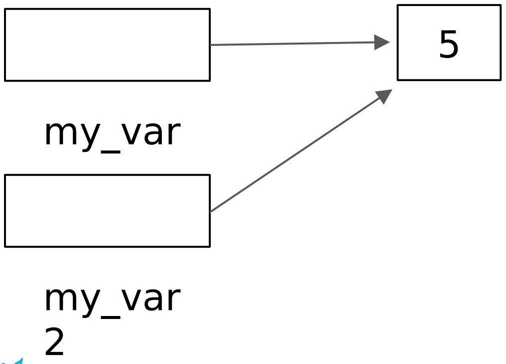
- In the first statement, `my_var` is assigned the memory address of an integer object.



Garbage Collection

```
my_var = 5
my_var2 = my_var
my_var = 10
print(my_var2)
```

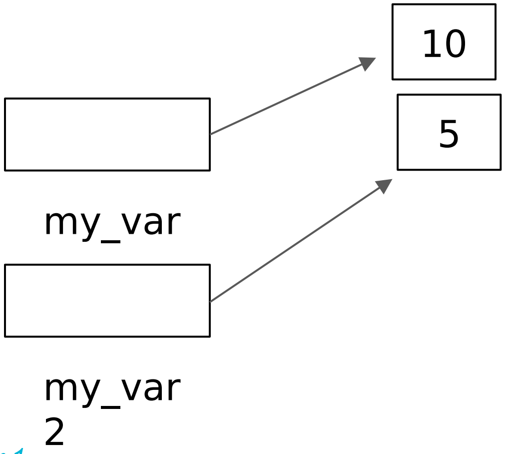
- In the second statement, `my_var2` is assigned the same memory address as `my_var`.
  - i.e. they both point to the same object.



Immutability

```
my_var = 5
my_var2 = my_var
my_var = 10
print(my_var2)
```

- Integer objects are **immutable**: once an int object is created, it cannot be changed.
- Therefore, the third statement *cannot* change the object `my_var` points to.
  - Rather, Python creates a new object containing 10 and points `my_var` to it.



- Because at least one variable still points to the first integer object we created, it does not get garbage collected.

Immutability

- The following types of objects are immutable in Python:
  - int, float, decimal
  - bool
  - string
  - tuple, range

Lists

```
my_list = [1, 2, 3]
my_list2 = my_list
my_list[1] = 5
for i in my_list2:
   print(i, end=" ")
```

- What will this code print?

Mutability

```
my_list = [1, 2, 3]
my_list2 = my_list
my_list[1] = 5                      # update original list
for i in my_list2:
   print(i, end=" ")
```

- What will this code print? 1, 5, 3
- Lists (like many other objects in Python) are **mutable**.
- Statement 3 does <u>not</u> create a new list.
  - Rather, it *updates* index 1 of the list `my_list` points to.
  - This happens to be the same object `my_list2` points to.
- **Note**: Most objects in Python are mutable.

Function Basics

A **function** is a reusable piece of code that can be called from anywhere else in our program. Some benefits of functions:

- They allow us to write blocks of code to use again and again.
  - Suppose we write a function to add two numbers and return the result. We never have to write another function that does that.
  - A program may perform the same action over and over. A function allows us to write the code to do this action a single time.
- We can split large programs into smaller, more manageable pieces.
  - Instead of having a single `main` function with 1000 lines of code, it is easier to have a much shorter `main` function that calls other functions.

Creating New Functions

- To define a function in Python, we first write its **header**.
- To do this, we use the `def` keyword before the function's name, followed by a list of 0 or more parameters, followed by a colon.
- The following header is for a function named `my_function` with no parameters:

```
def my_function():
        # function body goes here
```

- The function **body** is made up of one or more statements that will execute each time the function is called.
- Any code that is indented further than the function header is part of its body.

Function Basics

- Let’s write and call a function that takes one parameter.
- The argument passed to it is cubed and the result is printed out:

Function Basics

- Let's write and call a function that takes one parameter.
- The argument passed to it is cubed and the result is printed out:

```
def cube(x):
    print(x ** 3)


cube(3)                   # Call function cube, passing 3 as an argument
```

- In this code, when we call `cube`, we pass the integer literal 3 as an argument.
- When the function starts executing, the value 3 is assigned to the parameter `x`.
- Inside the function, we cube the value of `x` and pass the result to the `print` function.
- This code will print 27.

Function Basics

```python
def increase(x):
    x = x + 1


y = 3
increase(y)
print(y)
```

What will this program print?

Function Basics

```python
def increase(x):
    x = x + 1                          # Creates a new object!


y = 3
increase(y)
print(y)
```

What will this program print? 3

- Python passes arguments by **reference**.
  - This means a parameter points to the same object as its argument.
- **However**, recall that integers are immutable!
  - A new int object is created and assigned to *x* when we try to change its value.
- Therefore, *y* still references the object containing 3 when the `increase` function is finished executing.

Function Basics

```
def increase_all(a_list):
    for i in range(len(a_list)):
        a_list[i] += 1

my_list = [1, 2, 3]
increase_all(my_list)
print(my_list)
```

What will this program print?

Function Basics

```
def increase_all(a_list):
    for i in range(len(a_list)):
        a_list[i] += 1                    # Does not create a new List object

my_list = [1, 2, 3]
increase_all(my_list)
print(my_list)
```

What will this program print? [2, 3, 4]

- Because arguments are passed by reference, `a_list` references the same object as `my_list` when `increase_all` is executing.
- Because `List` objects are mutable, when we update `a_list` in `increase_all`, a new `List` is *not* created.
    - Any change to `a_list` updates the object referenced by `my_list`.

Creating New Functions

- Consider this code. What will print?

```python
def square(n):
   """Function Returns the square of n"""
   result = n ** 2


x = square(2)
print(x)
```

Creating New Functions

- Consider this code. What will print? **None**

```
def square(n):
    """Function Returns the square of n"""
    result = n ** 2


x = square(2)
print(x)
```

- This is a very common error in Python!
- `square` doesn't explicitly return anything.
    - However, all functions in Python return a `NoneType` object unless told to return something else, so the program executes without Error.
- In this example, a `NoneType` object is returned from `square` and assigned to *x*.

Creating New Functions

```python
def square(n):
    """Function Returns the square of n"""
    result = n ** 2
    return result

x = square(2)           # Assigns 4 to x
print(x)
y = square(5)           # Assigns 25 to y
print(y)
```

- In this example, we explicitly tell Python to return the value of the `result` variable.
  - The values we'd expect are now assigned to `x` and `y`.
- Unlike other languages, a Python program will not crash if we try to assign the return value of a function that doesn't return a value to a variable.
- It is important to make sure we place a `return` statement in functions that require it.

Creating New Functions

```python
first()

def first():
  print("Executing First")
```

- Will this run?

Creating New Functions

```
first()     # ERROR!

def first():
   print("Executing First")
```

- Will this run? **No**: A function cannot be called before it is compiled.
- A Python program is compiled from top to bottom.
    - Any code not inside a function is executed during this process.
- Since the call to `first` is not inside another function, it is immediately executed when reached. `first` has not been compiled yet, so this won't run.
- The following example runs properly:

```
def first():
   print("Executing First")

first()
```

Strings and Their Operations

- In Python, a **string** object (or `str` object) is an example of a **sequence**.
  - It is very similar to a List of single-character strings.
- For example, we can iterate over a string one character at a time:

```
my_str = "Hello"
for char in my_str:
   print(char, end=" ")   # Prints H e l l o
```

- We can also index into a string:

```
print(my_str[2])              # Prints l
```

Immutability

- Like integers, Strings in Python are **immutable**.
- A `str` object cannot be changed once it's created. For example, the following code:

```
my_string = "Bill"
my_string[0] = 'J'
```

- Will raise the following error:

```
TypeError: 'str' object does not support item assignment
```

- To make the desired change, we need to create a new `str` object and assign it to the `my_string` variable:

```
my_string = "Bill"
my_string = "Jill"
```

Slices

- The **slice** operator is used to obtain a substring of a string.
    - It looks similar to the subscript operator.
- What will the following code segments assign to `substring`?

```
my_string = "Cypress College"
substring = my_string[2:5]
```

```
my_string = "Cypress College"
substring = my_string[:7]
```

```
my_string = "Cypress College"
substring = my_string[:]
```

Slices

- The **slice** operator is used to obtain a substring of a string.
  - It looks similar to the subscript operator.
- What will the following code segments assign to `substring`?

```
my_string = "Cypress College"
substring = my_string[2:5]
```

pre

```
my_string = "Cypress College"
substring = my_string[:7]
```

Cypress

```
my_string = "Cypress College"
substring = my_string[:]
```

Cypress College

- The lower bound is inclusive and the upper bound is exclusive.
  - If we leave the lower bound out, it is assumed to be 0
  - If we leave the upper bound out, it is assumed to be length + 1

Slices

- What will this code print?

```
my_string = "Cypress College"
my_string[2:5]
print(my_string)
```

Slices

- What will this code print? **Cypress College**

```
my_string = "Cypress College"
my_string[2:5]
print(my_string)
```

- Since strings are immutable, *any* operation performed on a string actually returns a new string object.
    - Therefore, slicing does not change the original string.
- The result of slicing must always be assigned to a variable if we want to use it:

```
my_string = "Cypress College"
substring = my_string[2:5]
print(substring)
```

Strings and Their Operations

- What will this code print?

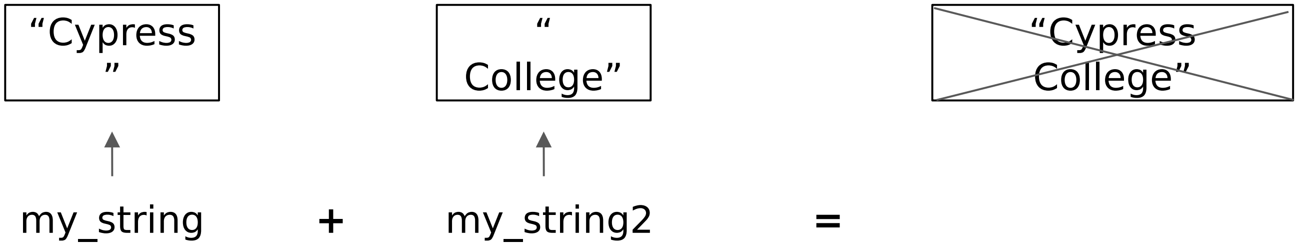
```
my_string = "Cypress"
my_string2 = " College"
my_string + my_string2
print(my_string)
```

Strings and Their Operations

- What will this code print? **Cypress**

```
my_string = "Cypress"
my_string2 = " College"
my_string + my_string2
print(my_string)
```

- When both operands of the + operator are strings, it performs concatenation on them.
  - However, the original string objects are not updated.
  - Rather, a third string object is created and returned
- In this example, the new object is not assigned to a variable and is garbage collected.
- When the call to `print` is executed, `my_string` still references a string object with the value “Cypress”.



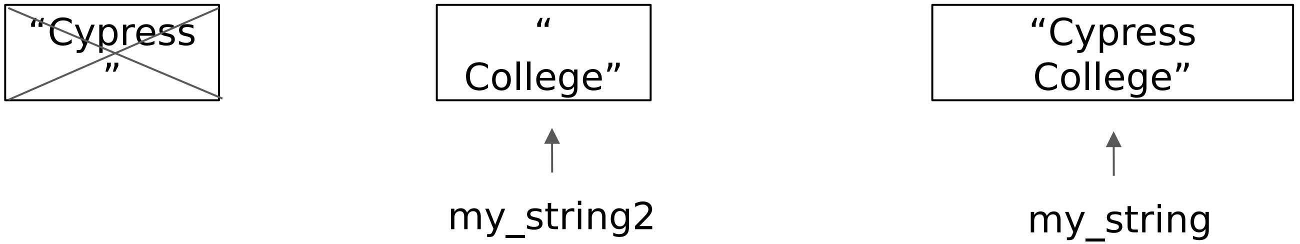
Strings and Their Operations

- To keep a concatenated string in memory, it needs to be referenced by a variable.

```
my_string = "Cypress"
my_string2 = " College"
my_string = my_string + my_string2
print(my_string)
```

- In this example, the right side of the third statement creates and returns a string object containing “Cypress College”
- It is is then assigned to `my_string`.

- Finally, the original object containing just “Cypress” is garbage collected:



Objects and Method Calls

- A **method** is similar to a function:
  - It may accept arguments.
  - It performs a task.
  - It returns a value.
- The main difference is that a method is always called on an object.
- We use the **dot operator** to specify which object to call a method on:

```
my_string = "Cypress"
my_string.isupper()
```

- This code calls the `isupper` method on the object referenced by `my_string`.
  - **Note**: A method can only be called on an object if that method is part of the class definition of the object's type.

Built-In Python Collections

- One of the most common Python collections is a **list**.
  - Like a string, a list is also an example of a sequence.
  - Unlike a string, a list is **mutable**.
- A `list` is a sequence of 0 or more Python objects of any type.
- To create an empty list and assign it to a variable, we use brackets:

```
my_list = []
```

- We can also create a `list` with elements already in it:

```
my_list = ["Hello", 15, "Jonathan"]
```

Built-In Python Collections

- What will this code do?

```
my_list = []
my_list[0] = 5
```

Built-In Python Collections

- What will this code do?

```
my_list = []
my_list[0] = 5
```

```
IndexError: list assignment index out of range
```

- Although a list grows as items are added to it, we cannot use the subscript operator to retrieve from or edit an index that doesn't exist in the list.
- Because `my_list` currently has no indexes, we cannot assign to its first (0th) index because it doesn't have one.

Built-In Python Collections

```
my_list = []
print(len(my_list))   # prints 0
my_list.append(5)
print(len(my_list))   # prints 1
my_list[0] = 7
print(my_list[0])     # prints 7
```

- In this code, we first create an empty list.
  - **Note**: When we pass a collection to the `len` function, the collection's size is returned.
- The `append` method adds a new index to the list. In this code `my_list` grows from 0 to 1 and the value '5' is added to the new index.
- In the final assignment statement, we can now use the subscript operator to assign a new value to index 0.

Other List Operations

- Lists have several other mutator methods, such as:
  - `sort`: Sorts the items in a list (as long as Python knows how to compare them)
  - `pop`: Removes the item at the end of the list.
  - `remove(value)`: Removes the given value from the list, or raises an Error if the value does not exist.
  - `insert(index, item)`: Adds the given item to the given index in the list.

String Tokenizing

```
my_string = "Python is cool"
```

- How can we easily tokenize this string?
  - Recall that when we **tokenize** a string, we break it into its individual elements (or **tokens**).
  - Each token is separated by a **delimiter**.
  - For this example, each token is a word and the delimiter is the space character.

String Tokenizing

```
my_string = "Python is cool"
tokens = my_string.split()
print(tokens)                      # Prints: ['Python', 'is', 'cool']
```

- The **split** method operates on a string and returns a list of tokens.
  - By default, it treats the space character as the delimiter.
- If we have a string with a different delimiter, we need to pass that character (or characters) to split:

```
my_string = "Python,is,cool"
tokens = my_string.split(',')
print(tokens)                      # Prints: ['Python', 'is', 'cool']
```

The join Method

- `join` is a `str` method which accepts a list of words as an argument and concatenates them into a single return string.
- In the return string, each string from the argument list is separated by the string of the calling object:

```
my_list = ["Python", "is", "cool"]
first_string = " "
print(first_string.join(my_list))      # Each word separated by a space: Prints "Python is cool"

second_string = "Test"
print(second_string.join(my_list))     # Each word separated by "Test": Prints "PythonTestisTestcool"
```

Looping Over Sequences

- The `for` loop is used to iterate over items in a sequence (such as a string or a list).
- How can we use a `for` loop to print each element in `test_list`?

```
test_list = [67, 100, 22]
```

Looping Over Sequences

- The `for` loop is used to iterate over items in a sequence (such as a string or a list).
- How can we use a `for` loop to print each element in `test_list`?

```
test_list = [67, 100, 22]
for score in test_list:
   print(score)
```

- In each iteration of the loop, `score` is assigned the next element in the list.
  - i.e. in the first iteration, it is assigned 67, 100 in the second iteration, and 22 in the final iteration.
  - `score` is called the **iteration variable**. We can name an iteration variable with any valid Python identifier.

Recursive Functions

```
def print_stars(n):
    if n == 1:
        print("*")
    else:
        print("*", end="")
        print_stars(n - 1)
```

- What does this function do?

Recursive Functions

```
def print_stars(n):
    if n == 1:
        print("*")
    else:
        print("*", end="")
        print_stars(n - 1)
```

Base Case

Recursive Step

- What does this function do? Recursively prints *n* stars.
- A **recursive function** is a function that calls itself. There are two parts:
    - The **base case**: Perform some action (optional) and stop recursion.
    - The **recursive step**: This occurs in every level of recursion except the final one. Perform some action and call the function again.
        - Note: In each recursive call, the input value *n* should be updated to move us closer to the **base case**.

Recursive Functions

```
def print_stars(n):
    if n == 1:
        print("*")
    else:
        print("*", end="")
        print_stars(n - 1)
```

Suppose we call `print_stars(4)`. The following steps occur:

- $n = 4$: prints one star and calls `print_stars(3)`
    - $n = 3$: prints one star and calls `print_stars(2)`
        - $n = 2$: prints one star and calls `print_stars(1)`
            - $n = 1$: prints one star.
- As can be seen from these steps, four stars in total will print.

Recursive Functions

```
def print_stars(n):
    print("*", end="")
    print_stars(n - 1)
```

- What is wrong with this recursive function?

Recursive Functions

```
def print_stars(n):
    print("*", end="")
    print_stars(n - 1)
```

- What is wrong with this recursive function?
- It has no base case!
  - When n is 1, we print a star and then call print_stars(0).
  - When n is 0, we print a star and call print_stars(-1).
  - etc.
- Recursion will never stop, until we use so much memory that the program crashes.

Recursive Functions

```
def print_stars(n):
    if n == 1:
        print("*")
    else:
        print("*", end="")
        print_stars(n)
```

- What is wrong with this recursive function?

Recursive Functions

```
def print_stars(n):
    if n == 1:
        print("*")
    else:
        print("*", end="")
        print_stars(n)              # ERROR
```

- What is wrong with this recursive function?
- In the recursive step, we do not pass a smaller value when we call `print_stars`. If a value other than 1 is passed, this function will never end!
- **Remember**: Each recursive case should perform one small operation and then call the function in such a way that we move one step closer to the base case.

Recursive Functions

To define a recursive function, we follow these steps:

- Define the **base case**, which performs the final step.
- Define the **recursive case**, which performs a single step and then calls the same function, passing an input that brings us one step closer to the base case.

Recursive Functions

- Let's write another recursive function.
- This function should take an integer as a parameter. It should then print out that number and each positive integer lower than it.
- How should this be written?
  - Remember, it should be written *recursively*.

Recursive Functions

- Let's write another recursive function.
- This function should take an integer as a parameter. It should then print out that number and each positive integer lower than it.
- How should this be written?
  - Remember, it should be written *recursively*.

```
def print_nums(n):
    if n == 1:
        # Base case, since 1 is the smallest positive integer
        print(1)
    else:
        # Recursive case: Print n and call print_nums again with a value one lower
        print(n, end=" ")
        print_nums(n - 1)
```

Recursive Functions that Return Data

- Suppose we want to write a recursive function that calculates n!
    - Recall that the factorial of an integer is that integer multiplied by every integer lower than it.
    - i.e. 7! is 7 * 6 * 5 * 4 * 3 * 2 * 1
        - Thinking of the problem in a recursive way:
            - 7! = 7 * 6!
                - 6! = 6 * 5!
                    - 5! = 5 * 4!
                        - etc

- What could a recursive case look like for a function that calculates this?
- What is the base case?

Recursive Functions that Return Data

- What could a recursive case look like for a function that calculates this?
  - For the recursive case, we want to multiply *n* by the factorial of *n* - 1
- What is the base case?
  - The base case is when *n* = 1: The factorial of 1 is simply 1, so we return 1:

```
def factorial(n):
   if n == 1:
       return 1
   else:
       return n * factorial(n - 1)


def main():
   print(factorial(7))
```

Recursive Functions that Return Data

- Let's try $n = 3$

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(3))
```

factorial(3)

n = 3 Multiply 3 by the value returned by factorial(2)

- In the top-level call to `factorial`, we are not at a base case, so we perform the recursive case.
- We return the result of a multiplication expression.
  - The right operand of this expression is a function call, which must be evaluated before we multiply it by n and return it.

Recursive Functions that Return Data

- Let's try $n = 3$

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(3))
```

factorial(3)

n = 3

factorial(2)

n = 2

Multiply 2 by the value returned by factorial(1)

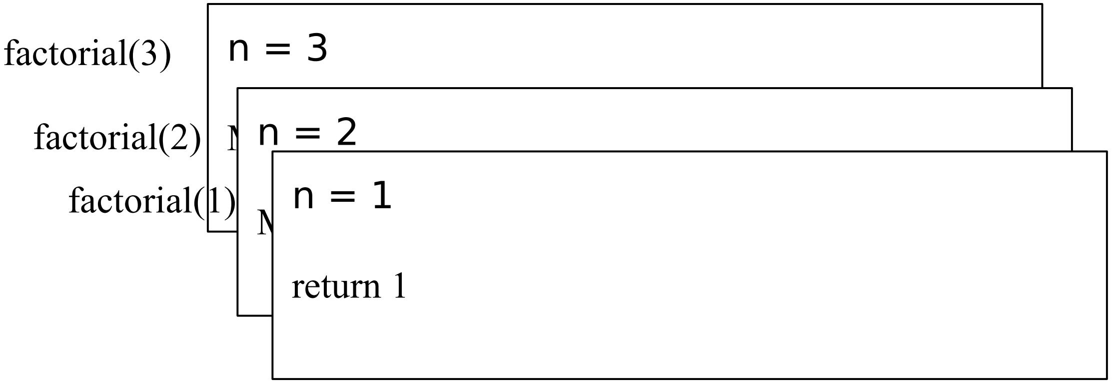
- In the next recursive level, we will multiply 2 by the result of calling `factorial(1)`

Recursive Functions that Return Data

- Let's try $n = 3$

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(3))
```



- In the next recursive level, we are at the base case. Return 1.

Recursive Functions that Return Data

- Let's try $n = 3$

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(3))
```

factorial(3)

n = 3

factorial(2)

n = 2

Multiply 2 by 1 and return it

- In the previous level of recursion, we multiply 2 by the value returned by `factorial(1)`, which is 1.
- We then return the result to the previous level of recursion.

Recursive Functions that Return Data

- Let's try *n* = 3

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(3))
```

factorial(3)

n = 3 Multiply 3 by 2 and return it

- Back in the top level of recursion, we multiply 3 by the value returned by `factorial(2)`, which is 2.
- Since this was the first level of recursion, the function is finished:
  - We return the result, which is 6.
  - The `print` function then prints out '6'.

Catching Exceptions

```
name = input("Enter your name: ")
age = input("Enter your age: ")
print(f'Hello, {name}. You have been alive approximately {age * 365} days')
```

- Suppose the user enters “Jonathan” and “37”.
- What will this program output?

Catching Exceptions

```
name = input("Enter your name: ")
age = input("Enter your age: ")
print(f'Hello, {name}. You have been alive approximately {age * 365} days')
```

- Suppose the user enters “Jonathan” and “37”.
- What will this program output?
    - “Hello Jonathan. You have been alive approximately 373737373737373737…(etc) days”.
- The `input` function always returns a string.
- When we multiply a string by a number *n*, that string is repeated *n* times.
- In this case, 37 is repeated 365 times.

Catching Exceptions

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print(f'Hello, {name}. You have been alive approximately {age * 365} days')
```

- In this program, we now cast the user's input to an integer.
- Can anything go wrong now?

Catching Exceptions

```python
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print(f'Hello, {name}. You have been alive approximately {age * 365} days')
```

- In this program, we now cast the user’s input to an integer.
- Can anything go wrong now?
  - If the user enters a string with at least one non-numeric character, an Error (or exception) occurs!
    - The argument to `int` <u>must</u> be a string with only numeric characters.
  - If this exception is not caught, the program will crash.

Catching Exceptions

- A try-except statement allows a program to **catch** an exception and perform a recovery operation.
- Rather than crashing when a runtime error occurs, we can inform the user of the error and keep the program running.
- This type of statement has the following syntax:

```
try:
   <statements>
except:
   <statements>
```

Catching Exceptions

```
try:
   <statements>
except <exception type>:
   <statements>
```

- When a try-except statement executes, the statements in the `try` clause are executed.
- If any of these statements raises an error, control immediately transfers to the `except` clause.

Catching Exceptions

```
name = input("Enter your name: ")
try:
   age = int(input("Enter your age: "))
   print(f'Hello, {name}. You have been alive approximately {age * 365} days')
except ValueError:
   print("You didn't enter a number!")
```

- While this code now avoids a crash, is there any way to improve it?

Catching Exceptions

```
name = input("Enter your name: ")
try:
    age = int(input("Enter your age: "))
    print(f'Hello, {name}. You have been alive approximately {age * 365} days')
except ValueError:
    print("You didn't enter a number!")
```

- While this code now avoids a crash, is there any way to improve it?
- We can change it to continually ask the user for input until they enter a number.
    - In fact, let’s write a helper function that retrieves an integer from the user.

Catching Exceptions

```
def get_integer(prompt):
    """ Prompts the user for an integer and returns it if it is valid.
    Otherwise, prints an error message and repeats the process """
    input_string = input(prompt)
    try:
        number = int(input_string)
        return number
    except ValueError:
        print(f'Incorrect input: {input_string}')
        print("Please enter a number.")
        return get_integer(prompt)      # recursively call get_integer.
```

- This recursive function returns the user's input if it is a properly formatted integer.
- Otherwise, it returns the result of calling `get_integer` recursively.
    - This gives the user another chance to enter a correct value.

Catching Exceptions

```
name = input("Enter your name: ")
age = get_integer("Enter your age: ")
employee_ID = get_integer("Enter your id: ")
```

- Helper functions should be written to be flexible.
- We made `get_integer` flexible by allowing the client to pass a prompt string. The function can therefore be used any time we need to retrieve an integer from the user.