CSCI A201/A597 - Tools For Computing: Python

reminder:

turn on all notifications and set them to "right away" in your IU Canvas account, so that you receive (to your email account, and as push notifications on your mobile device Canvas app) any announcements, grade updates, comments, etc... as soon as they're posted!

Here's where you find it in Canvas, from a desktop web browser:

(the mobile version may look different)
Lecture 04 – Topics

→ let's start with a review (since we had no lecture on Labor Day last week!) →

1st topic (review):
functions & methods (calling them)

2nd topic (review):
functions (defining our own functions)

3rd topic:

**positional** parameters, **keyword** arguments,
and **default parameter values**

4th topic:
scopes: local & global
Review of Python Terminology

**print()** is a Python **Function** that can display a **string** (actually, it can display any **expression**)

**String**: Sequence of characters

**Function**: a block of organized programming code that can be executed by calling its name (in this case, **print()**). A function may receive values, and do some work with those values.

**Argument**: Value passed to a function

```python
print("Game Over")
```
Review of Python Terminology (continued)

some more terminology:

**Statement:** a single unit in programming language that performs some action.

**Expression:** Something which has a value or that can be evaluated to a single value

"Game Over"

7 + 2

**Code:** Sequence of programming statements, function calls, etc.
Review of Python Terminology (continued)

**Syntax highlighting**: Displaying programming code in different colors or fonts, according to the category of each item

**Errors**

Computers take everything literally

```python
primt ("Game Over") produces an Error Message:
NameError: name 'primt' is not defined
```

**Name error**: in this case, a function with the name 'primt' could not be found by Python, and Python alerts us of this!

**Bug**: Error in programming code
Photo # NH 96566-KN  First Computer "Bug", 1945

0800  Started
1000  Stopped

13° C (93) MP-MR  2.13678479 (20) 4.615925085 (2)

(03) PRO = 2.13678479
Conv. 2.13678479

Relays 6-2 in 033 failed special speed test

1800  Started Cosine Tape (Sine check)

1525  Started Multi Adder Test

1545  First actual case of bug being found.

1600  Antenna started.

1700  Closed down.
Review: Built-in Functions

We have seen that Python comes with useful built-in code, for example, in our code we may call some of these functions:

input()    # gets input from user (has side effect!)
print()    # displays output (has side-effect!)
int()      # converts data to integer
str()      # converts data to string
float()    # converts data to floating-point

These are just some of the functions that are built-in Python. For a complete list, see:
http://docs.python.org/3/library/functions.html
Review: Built-in Functions

in Python (as in other programming languages), there is input to the function and there's output from the function:

\[
\text{outputFromFunction} = \text{functionName}(\text{inputToFunction})
\]

The input may be zero, one, or more arguments, we call arguments the values we pass to a function when we call that function.

Some functions may also cause side effects, e.g.:

\[
\text{print("who's the output now?")}
\]

What is the "side effect" of the above print() statement? (answer this on your own: does print() produce an output?)

if you don't remember side effects, re-read Reading Assignment 03, specifically:

https://homes.sice.indiana.edu/classes/fall2018/csci/a201-mitja/2018/read/read03.html#purefunction
Review: Objects and Methods

(the words method and function are often used interchangeably by programmers)

Let’s use method to refer to a specific type of function in Python – one that’s tied to an object.

What is an object?

In Python, an object is defined as:
"data with both: value(and type) and defined behavior (methods)"

All Python variables are actually objects.
For example strings, numbers, etc.

We call a method like this:

objectName methodName()
Review: Calling an object's Methods

We call a method like this:

```
objectName.methodName()
```

**dot notation:**

Use *variable name* for object, followed by *dot*, followed by *method name* and parentheses

```
objectName.methodName()
someString.upper()
"abc".upper()  # ← note: you can also call the method of a *value*
```

By comparison, functions like `input()` are called on their own:

```
user_name = input("Please enter your name: ")
```
Review: Using String Methods

**String methods** allow you to do many things, including:

- Create new strings from old ones
- Create a string that’s an uppercase version of the original
- Create a new string from the original, based on letter or word substitutions

If you don't remember *methods*, re-read material/notes in Programming Assignment 02, specifically at:

https://homes.sice.indiana.edu/classes/fall2018/csci/a201-mitja/2018/assignments/pa02.html#stringmethods
Review: String Methods:

someText = "I like Python."
someText.upper() → upper() returns a version of the string where everything is in capital letters
   "I LIKE PYTHON."
someText.lower() → lower() is similar, for lowercase letters.
   "i like python."

These two can be useful to test user input, e.g:
   you may compare a variable someText.upper()
to a value obtained from input, e.g.: 
   userName = input("provide your username")
   that can be then used as "all capital letters" thus: userName.upper()
   but we haven't yet learn how to compare... that's coming soon!
Review: String Methods (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper()</td>
<td>Returns the uppercase version of the string.</td>
</tr>
<tr>
<td>lower()</td>
<td>Returns the lowercase version of the string.</td>
</tr>
<tr>
<td>swapcase()</td>
<td>Returns a new string where the case of each letter is switched. Uppercase becomes lowercase and lowercase becomes uppercase.</td>
</tr>
<tr>
<td>capitalize()</td>
<td>Returns a new string where the first letter is capitalized and the rest are lowercase.</td>
</tr>
<tr>
<td>title()</td>
<td>Returns a new string where the first letter of each word is capitalized and all others are lowercase.</td>
</tr>
<tr>
<td>strip()</td>
<td>Returns a string where all the white space (tabs, spaces, and newlines) at the beginning and end is removed.</td>
</tr>
<tr>
<td>replace(old, new [,max])</td>
<td>Returns a string where occurrences of the string old are replaced with the string new. The optional max limits the number of replacements.</td>
</tr>
</tbody>
</table>

optional parameter
Review: Working with Numbers

We can work with numbers as easily as with strings

Numeric types

**Integers**: Numbers without a fractional part

1, 0, 27, -100

**Floats (Floating-Point Numbers)**: Numbers with a fractional part (even if it’s .0)

2.376, -99.1, 1.0
# Review: Mathematical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Evaluates To</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>7 + 3</td>
<td>10</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>7 – 3</td>
<td>4</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>7 * 3</td>
<td>21</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>7 / 3</td>
<td>2.33333333...</td>
</tr>
<tr>
<td>//</td>
<td>Integer Division</td>
<td>7 // 3</td>
<td>2</td>
</tr>
<tr>
<td>%</td>
<td>Modulus (remainder)</td>
<td>7 % 3</td>
<td>1</td>
</tr>
</tbody>
</table>
Review: Division and Modulus

To convert a number into a decimal (float), we can use the `float()` function OR multiply/divide by a decimal number:

- `float(4)` \rightarrow 4.0
- \((9 \times 3.0) / 2\) \rightarrow 13.5

**Modulus** gives the remainder under division

the mathematical expression "10 \(\text{mod}\) 3"

means "what is the integer remainder when 10 is divided by 3?" \rightarrow 1

\((10 \text{ mod } 2)\) is 0, because 2 evenly divides 10.

In Python, we use \(\%\) for mod. \(10 \% 2 \rightarrow 0\)
Review: "Augmented Assignment Statements"

Often, we assign a new value to a variable, based on its original value – this can be also done with Augmented Assignment Statements:

Original: `score = score + 1`
Augmented: `score += 1`

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Is Equivalent To</th>
</tr>
</thead>
<tbody>
<tr>
<td>*=</td>
<td><code>x *= 5</code></td>
<td><code>x = x * 5</code></td>
</tr>
<tr>
<td>/=</td>
<td><code>x /= 5</code></td>
<td><code>x = x / 2</code></td>
</tr>
<tr>
<td>%=</td>
<td><code>x %= 5</code></td>
<td><code>x = x % 5</code></td>
</tr>
<tr>
<td>+=</td>
<td><code>x += 5</code></td>
<td><code>x = x + 5</code></td>
</tr>
<tr>
<td>-=</td>
<td><code>x -= 5</code></td>
<td><code>x = x - 5</code></td>
</tr>
</tbody>
</table>
Review: "Augmented Assignment Statements"

Notes about augmented assignment operators:

1. *they are not compulsory:*
   it is perfectly fine to use `score = score + 1` instead of `score += 1` (it’s a matter of taste)

2. *the order in which they are typed matters:*
   `score += 1` is *not the same* as `score += 1`
   *(give it a try... see what happens!)*

3. *use them carefully!*
Review: ...and what happens when you divide by 0?

Beware! If you divide by 0, this may happen to your program...
Review: Boolean Conditions

A **Boolean condition** is anything that is either **True** or **False**.
Here are some statements that can be either True or False.
Can you evaluate their True/False value?
If yes, what is it?
If not, what is missing for you to be able to determine the True/False value?

- a variable has the value “test” in it
- $10 > 5$
- a variable’s value is not 0
## Review: Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Example</th>
<th>Truth Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equals</td>
<td>5 == 6</td>
<td>False</td>
</tr>
<tr>
<td>!=</td>
<td>Not Equal</td>
<td>5 != 6</td>
<td>True</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater Than</td>
<td>5 &gt; 6</td>
<td>False</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater Than or Equal</td>
<td>5 &gt;= 6</td>
<td>False</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less Than</td>
<td>5 &lt; 6</td>
<td>True</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less Than or Equal</td>
<td>5 &lt;= 6</td>
<td>True</td>
</tr>
</tbody>
</table>

Warning! It’s easy to confuse = and ==. They mean very different things....
Review: Writing Your Own Functions!

- goal: write your own functions
- pass values to your functions through arguments
- accept values into your functions through parameters
- return information from your functions through return values
Review: Creating-Writing-Defining Functions

• in our A201/A597 course, we'll assume that these three terms mean the same: creating functions == writing functions == defining functions

• you can define functions of your own

• writing functions allows you to break up code into manageable chunks (vice versa, programs that are just an unbroken long series of instructions are harder to write, understand, and maintain)

• just like Python built-in functions, the functions you write should do one job well
Review: Defining a Function

- Functions make programs easier to read, write and maintain.

- Function definition: Code that defines what a new function does.

- Function header: First line of a function definition.

- Give your function a name that conveys what it does or produces.
Review: Documenting a Function

def doubleThis(pValue): # doubleThis() is a function that has one parameter named pValue
    """ this is a docString: doubleThis() returns the input value, multiplied by two """
    # pValue is a parameter that is used within the doubleThis() function:
    # the pValue parameter receives whichever argument value passed by the
    # code that is calling this function.
    return 2 * pValue    # the doubleThis() function returns the computed value

da Docstring is a String that documents a function

Docstrings are:
    triple-quoted strings.
    must be the first line in your function
    not required in Python, but a good idea – and mandatory in A201/A597
Pop up as interactive documentation in IDLE, e.g.:

```python
>>> myResult = doubleThis
(pValue)
this is a docString: doubleThis() returns the input value, multiplied by two
```
Review: Our Goals when Writing Functions!

Write your own **functions**
Passing values to your functions through **arguments**
Accept values into your functions through **parameters**
Return information from your functions through **return values**
Work with **global variables** and constants
Review: Creating Functions

in our class,
  we'll assume that these three terms mean the same:
  
  creating functions == writing functions == defining functions

you can define functions of your own
functions let you to break up code into manageable chunks
  (programs that are a long series of instructions are
  harder to write, understand, and maintain)
just like Python built-in functions,
  the functions you write should do one job well
Review: Defining a Function

```
def give_me_five():  # give_me_five() is a function with no parameters
    lFive = 5          # lFive is a local variable - it exists only inside the function
    return lFive      # the give_me_five() function returns a value
```

Functions make programs easier to read, write and maintain

Function **definition**: Code that defines what a new function does

Function **header**: First line of a function definition

Give your function a **name** that conveys what it does or produces
**Review: Documenting a Function**

```python
def doubleThis(pValue):
   # doubleThis() is a function that has one parameter named pValue
   """ this is a docString: doubleThis() returns the input value, multiplied by two """
   # pValue is a parameter that is used within the doubleThis() function:
   # the pValue parameter receives whichever argument value passed by the
   # code that is calling this function.
   return 2 * pValue  # the doubleThis() function returns the computed value
```

**Docstring:** a String that documents a function

**Docstrings are:**

- triple-quoted strings.
- must be the first line in your function
- not required in Python, but a good idea – and mandatory in A201/A597

Pop up as interactive documentation in IDLE, e.g.:

```python
>>> myResult = doubleThis(5)
5
```

```
(pValue)
this is a docString: doubleThis() returns the input value, multiplied by two
```
Review: Using Parameters and Return Values

Just as with built-in functions

Your functions can get values

  arguments: what you pass to the function
  parameters: where the function places what you pass to it

Your functions can return values
Review: Receiving Information through Parameters

**Parameter:** A *variable name* inside the parentheses of a function header that can receive a value

**Argument:** A *value* passed to a parameter

Parameters must receive values; otherwise... error!

Multiple parameters can be listed, separated by commas

Here's a not-so-good example of a function that accepts a parameter:

```python
def instructions(pString):
    """Prepare game instructions."""
    # <-- this function provides a docString
    print( pString )  # WARNING!!! print statements usually not to be used in functions!
    # the instructions() function returns nothing.
    # this is where our main code calls the instructions() function,
    # passing the string "welcome to the game" as argument to the instructions() function:
instructions("welcome to the game")
```

(note: unless the main purpose of a function is to print something, it's best to avoid print statements inside functions, other than for debugging)
Review: Returning Information through Return Values

```python
def give_me_five():
    lFive = 5
    return lFive
```

Return value: a value returned by a function
`return` statement returns values from a function
`return` statement ends function call
Can return more than one value from a function -- list all the values in `return` statement, separated by commas
Sample call:

```
number = give_me_five()
```
Summarizing the Review: defining and calling functions, some simple examples

```
# give_me_five() is a function with no parameters
lFive = 5
return lFive

# doubleThis() is a function that has one parameter named pValue
""" this is a docString: doubleThis() returns the input value, multiplied by two """
# pValue is a parameter that is used within the doubleThis() function:
# the pValue parameter receives whichever argument value passed by the
# code that is calling this function.
return 2 * pValue

# instructions() is a function with one parameter
"""Prepare game instructions."""

print( pString )

# the instructions() function returns nothing.

# the main program begins here

# this is where our main code calls the instructions() function,
# passing the string "welcome to the game" as argument to the instructions() function:
instructions("welcome to the game")

print("this is going to print the output of give_me_five() : ", give_me_five())

myDoubleValue = doubleThis(22)
# in the above line, 22 is the argument passed to the doubleThis() function
print (myDoubleValue)

print("now testing doubleThis() on value 1. The result is:", doubleThis(1) )
print("now testing doubleThis() on value -99. The result is:", doubleThis(-99) )

myInstructions = instructions()

print (myInstructions)
```
Lecture 04 – Topics

1st topic (review):
functions (calling them)

2nd topic (review):
functions (defining them)

3rd topic:
positiona l parameters, keyword arguments,
and default parameter values

4th topic:
scopes: local & global
Positional Parameters with Positional Arguments

def birthday1(name, age):
    print("Happy birthday,", name, "!", "You’re", age, ". ")

How do we use positional parameters when defining our own function?
That's actually nothing new: that's what we have done every time we've defined a function so far: we write a list of parameters in the function header.
For example, in the above function definition for birthday1(), name and age are positional parameters

Note: positional parameters are the "regular" way of defining parameters in Python, and are the only kind of parameters that we have used so far (i.e. before this Lecture 04).
Positional Parameters and Positional Arguments (continued)

since positional arguments are just a sequence of argument values in a function call...

...with positional parameters and **positional arguments**, parameters get their values based on the order of the argument values sent when calling the function.

As we see from the above example use, we have to be careful about parameter positions!

Note: **positional arguments** are the "regular" way of writing arguments in Python, and are the only kind of arguments that we have used so far (before this Lecture 04)
without changing anything to our function definition, we can use **keyword arguments** when calling the function, i.e.:

**keyword arguments** are arguments that get passed to specific parameters using the parameter name(s) (parameter names can be found in the function definition, or also by just looking at the "tooltip")
Positional Parameters and Keyword Arguments

Parameter name(s) for any function can be found out in several ways:

1. If you wrote that function, look at the parameters you wrote in the function definition.

2. You can also look at the "tooltip":

3. In the Python shell, ask for `help()` about the function, which also shows the function's Docstring:
Using Keyword Arguments and Default Parameter Values

so far, we've seen that
we can pass values as arguments to specific parameters

```python
>>> birthday1("Jackson", 1)
Happy birthday, Jackson! I hear you’re 1 today.
```

but we can also define our function with **parameters** having **default values**, i.e. values that will be assigned to parameters if we pass no value as argument to the function!

* e.g. notice the difference between `birthday1()` and `birthday2()` function definitions here:
Default Parameter Values

A **default parameter** value is:
A value that a parameter gets if **no argument value** is passed to it:

```python
def birthday1(name, age):
    """prints out a birthday greeting (side effect!)
    String, Number -> None"
    print ("Happy birthday, " + str(name) + "! I hear you’re " + str(age) + " today. ")

def birthday2(name = "Jackson", age = 1):
    print ("Happy birthday, " + str(name) + "! I hear you’re " + str(age) + " today. ")
```

A **default parameter** value is:
A value that a parameter gets if **no argument value** is passed to it:

```python
>>> birthday1()
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    birthday1()
TypeError: birthday1() missing 2 required positional arguments: 'name' and 'age'
>>> birthday2()
Happy birthday, Jackson! I hear you’re 1 today.
```
Default Parameter Values (continued)

with the same function definition:

```python
def birthday2(name = "Jackson", age = 1):
    print ("Happy birthday, " + str(name) + "! I hear you’re " + str(age) + " today. ")
```

a few more example runs:

```python
>>> birthday2()
Happy birthday, Jackson! I hear you’re 1 today.
>>> birthday2(name = "Katherine")
Happy birthday, Katherine! I hear you’re 1 today.
>>> birthday2(age = 12)
Happy birthday, Jackson! I hear you’re 12 today.
>>> birthday2(name = "Katherine", age = 12)
Happy birthday, Katherine! I hear you’re 12 today.
>>> birthday2("Katherine", 12)
Happy birthday, Katherine! I hear you’re 12 today.
>>> birthday2(12, "Katherine")
Happy birthday, 12! I hear you’re Katherine today.
>>> 
>>> 
```
again: **keyword arguments & default parameter values**

source code for the "Birthday Wishes" program

• shows how to write **default parameter values** in function definitions (here shown when defining the `birthday1()` function)

• shows how to use **keyword arguments** when calling a function (here shown when calling the `birthday2()` function)
4th topic: Scopes, local & global

Scopes defined:

Scopes are different areas of a program that are separate from each other.

For example:
Every function in a program has its own scope.

Because of function having separate scopes, functions can't directly access each other's local variables.

(e.g. in this example, func1() can't access variable2, and func2() can't access variable1)

But in this example there are actually three separate scopes:
One for each function, one for the global scope.
global or local variables?

try running the Python script shown here.

what happens?

which ones are global variables?

which ones are local variables?

```python
# Global Reach
# Demonstrates global variables

def read_global():
    print("From inside the local scope of read_global(), value is:", value)

def shadow_global():
    value = -10
    print("From inside the local scope of shadow_global(), value is:", value)

def change_global():
    global value
    value = -10
    print("From inside the local scope of change_global(), value is:", value)

# main
# value is a global variable because we're in the global scope here
value = 10
print("In the global scope, value has been set to:", value, "\n")

read_global()
print("Back in the global scope, value is still:", value, "\n")

shadow_global()
print("Back in the global scope, value is still:", value, "\n")

change_global()
print("Back in the global scope, value has now changed to:", value)

input("\n\nPress the enter key to exit.")
```
Using Global Variables and Constants

**Global variables** are variables that can be accessed in any part of a program... or can they?

Try the example here:

```python
def cube(i):
    """cube(i) returns the cube of i
    number -> number""

    # compute the cube and place it into a local var:
    theResult = i * i * i
    # debugging here... printing out a local var:
    print("inside cube(), theResult = " + str(theResult))
    return theResult

# the "main" part of this Python script begins here:

# theResult is a "global" variable, seemingly
# unaffected by code inside functions:
theResult = 20

print("before calling cube(), theResult = " + str(theResult))
cube(6)
print("after calling cube(), theResult = " + str(theResult))

# note: the variable named "theResult" inside the function cube()
# is a local variable: it is local to the function cube()

# The variable named "theResult" in the "main" part of this script
# is a global variable, and it is a completely *different* variable
# from the local "theResult" variable inside the function cube().
# These two variables just happen to have the same name.
```

Global **constants** are constants that can be accessed in any part of a program...

(...however – are there any truly CONSTANT types in Python, i.e. variables whose values we can't ever change? We haven't considered that aspect yet)
Global variable:
A variable created in the global scope that can be accessed in any part of a program

Local variable:
A variable created in a scope other than the global scope that can't be accessed outside of its scope

We can read the value of a global variable from within any scope in your program... if we want to!
We must actually tell Python that we're using a global variable within a function.
Shadowing a Global Variable from Inside a Function

```python
def shadow_global():
    value = -10
    print("Inside shadow_global(), value is: ", value)

value = 10
shadow_global()
print("Back in global scope, value is still: ", value)
```

Shadow: to 'hide' access to a global variable from inside a scope, by creating a new local variable of the same name. The local variable takes precedence!

In general, it's not a good idea to shadow a global variable.
We can gain direct access to global variable with the Python keyword: `global`
Understanding When to Use Global Variables and Constants

Use of global variables can lead to confusion...

we suggest that you **limit** use of global variables!

Global constant:
Global variable *treated as* a constant.

Use of global constants, however, can make programs clearer. Just beware about what constant means in Python!