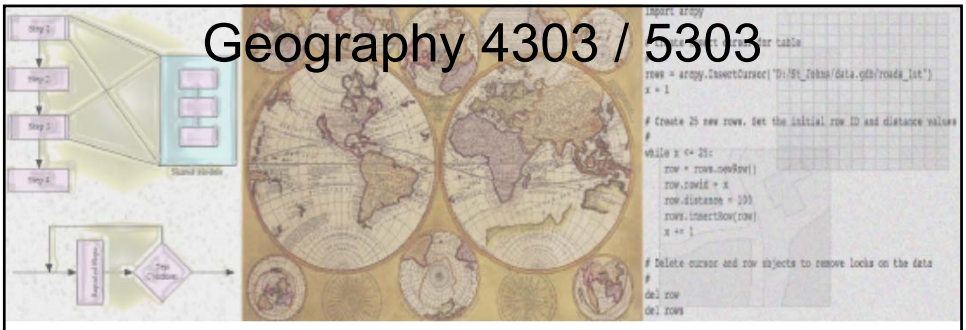


Geography 4303 / 5303

GIS PROGRAMMING FOR SPATIAL ANALYSIS

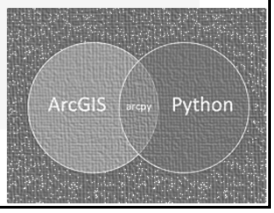
Class 04: Functions & Modularity in Python

Some Updates

- Project proposals due Feb 18 (to be presented on Mar 04)
- Each prospective project leader has to find at least ONE other person to work with
- Final project notes …
- Labs: Increasing challenge … independent work
- Approach (order of topics etc.; raster analysis)
- Testing and checking programs before sending
- Reusing YOUR OWN code for next labs …

Last Lecture / Last Week

- How to **write arcpy scripts** using tools from ArcToolbox as methods
- ***P***Managing, organizing, listing and manipulating spatial data with arcpy
- Data access - **Cursor** objects and **Geometry**; how to query, change and create spatial data geometry



Today ‘s Outline

- More on Python basics
- Program **reuse** and **modularity**
- We will talk about **functions** (Part I) and **modules** (Part II) – next time?
- The process of defining and using (calling) **user-specific functions**
- **User-specified modules**

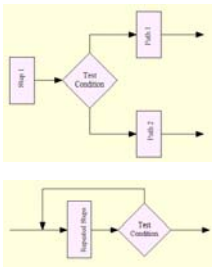
Learning Objectives

- You will learn what **functions** and **modules** are and why they are critical (means: “important“) elements in programming
- You will learn how to use and develop **user-specific** solutions as **reusable** pieces - modules and functions - to develop advanced **procedural** solutions

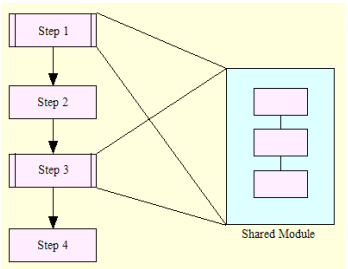
Recall: About Reuse

- You will learn what **functions** and **modules** are and why they are critical (means: “important“) elements in programming
- You will learn how to use and develop **user-specific** solutions as **reusable** pieces - modules and functions - to develop advanced **procedural** solutions

Not New: Modularity



- Getting access to and performing identical sequences of actions several times
- Actions are placed in a **module**, **sub-routine**, **procedure** or **function** that can be exectued from within the main program



Functions in General

- **Reusable** pieces of code
- Named sequence (or block) of statements to perform a desired **operation** or **action**
- Means: The **block** of statements has a “name”
- The block of statements can be **executed** using the name (repeated, anywhere)
- **Execution** = Calling the function

Example arcpy cl04_01, list.replace()

Recall: Built-in Functions

- Automatic availability of many „**built-in**“ functions without import of modules (Python core lib)
- List these functions by `dir(__builtins__)` in the interactive window

```
len()     - returns the length of Str
max()     - returns the maximum value of list
open()    - opens a file
round()   - rounds a number
dir()     - exported names within modules
pow()     - arg1 to power of arg2
```

Example built-in

How to Create and Call Functions in Python

- Using the **def** keyword
- => **identifier** (name of the function)
- => **parenteses** “()” to insert parameters
- => **Colon** “:”
- => **Block** of statements (body of function)
- Call the function: `tellMeHi()`

```
>>> def tellMeHi():
...     print "Hi, how are you?"
...
>>> tellMeHi()
Hi, how are you?
```

Multiply cl04_02, no par…

Functions and Parameters

- Functions take parameters (something is to be executed by utilizing them)
- **Parameters**: Variables whose values are defined before we call the function (outside)
- Specified in **parentheses** () when defining function and separated by **commas** (par1,par2,…,)

```
>>> def tellMeHi(a,b):
...     if a < b: print "The first entry ("+str(a)+") is the Minimum"
...     elif b < a: print "The second entry ("+str(b)+") is the Minimum"
...     else: "The two entries are equal"
...
```

Show again arcpy cl04_01 + custom. multiply cl04_02 Fct. (par)

Functions and Arguments

- **Arguments** are given in parentheses when we **call** the function
- Given in the **same order** as the parameters for defining the function (as literals or variables)
- The value of `arg1` is assigned to `par1` in the function definition

```
>>> def tellMeHi(a,b):
...     if a < b: print "The first entry ("+str(a)+") is the Minimum"
...     elif b < a: print "The second entry ("+str(b)+") is the Minimum"
...     else: "The two entries are equal"
...
>>> tellMeHi(4,6)
The first entry (4) is the Minimum
```

```
>>> i = 7
>>> j = 9
>>> tellMeHi(i,j)
The first entry (7) is the Minimum
```

Show calling arcpy cl04_01 + custom. multiply cl04_02 Fct. (arg)

Local Variables in Functions

- Variables declared **within a function** do not exist outside that function
- **Scope** of the variable: "**local** to the function"
- Determined by the block in which the variable is declared

```
>>> def tellMeHi(c):
...     print "The variable c (outside the function) is: ", c
...     c = 9
...     print "The local variable is: ", c
...
>>> c = 49
>>> tellMeHi(c)
The variable c (outside the funtion) is:  49
The local variable is:  9
>>> c
49
```

Global Variables in Functions

- If you want to assign to / change a variable inside your function so that this variable exists outside the function:

  `Global` statement

- Declare that the variable is global

```
>>> def tellMeHi():
...     global c
...     print "The variable c (outside the function) is: ", c
...     c = 2
...     print "The local variable is: ", c
...
>>> c = 45
>>> tellMeHi()
The variable c (outside the function) is:  45
The local variable is:  2
>>> c
2
```

Arguments with Default Values

- **"Optional"** parameters (**default** values if user does not provide a value)
- `func(parName = default)`
- At the end of par / arg lists (values are assigned by position)
- **Immutable**
- ArcPy?

```
>>> def tellMeHiAsOftenAsYouWant(varHi, times = 1):
...     print varHi * times
...
>>> tellMeHiAsOftenAsYouWant('Moin')
Moin
>>> tellMeHiAsOftenAsYouWant('Moin', 4)
MoinMoinMoinMoin
```

Change multiply

Keyword arguments

- To specify only some parameters, assign values by **naming** them when calling functions
- **Names** instead of **position - order** of arguments not important
- **Selective assignment** of values of interest (others have default values)

```
>>> def myFunc(a, b = 4, c = 2):
...     print 'a = ', a, ', b = ', b, ', c = ', c
...
>>> myFunc(45)
a =  45 , b =  4  c =  2
>>> myFunc(23, c = 32)
a =  23 , b =  4  c =  32
>>> myFunc(c = 32, a = 22, b = 100)
a =  22 , b =  100  c =  32
```

Caution

- Python, while a modern programming language, “suffers” from high degrees of flexibility
- For Example: Keep your **parameter lists** always in the right order…
- Even if this could be useful in some cases, using **keywords** can cause confusion and in other languages this is not even allowed

“Return” Statements

- “Return from a function” (break out)
- “Return a value from the function”
- No return = `Return None` (every function has at least a `Return None` statement)
- Return multiple values vs. multiple returns

```
>>> def myFunc(a, b):
...     if a < b:
...             return a
...     elif a > b:
...             return b
...     else:
...             return a+b
...
>>> myFunc(3,4)
3
>>> myFunc(4,4)
8
```

```
>>> def myFunc(a, b):
...     if a < b:
...             a = b
...     elif a > b:
...             b = 3
...     else:
...             a = 7
...
>>> myFunc(3,4)
>>>
>>> print myFunc(3,2)
None
```

Back to multiply; multiple returns

DocStrings in Functions

- **Documenting** your program (DocString from a function while program is running)
- Access to DocString with __doc__ attribute
- `Help()` is one example
- Conventions

```
>>> print myFunc.__doc__
This function returns the smaller value out of two values.

    The two values must be Integers. If not
    you have to use an alternative function.
```

```
def myFunc(a, b):
    '''This function returns the smaller value out of two values.

    The two values must be Integers. If not
    you have to use an alternative function.'''

    if a < b:
        return a
    elif a > b:
        return b
    else:
        print "There is no unique Minimum"
```

```
>>> help(myFunc)
Help on function myFunc in module __main__:

myFunc(a, b)
    This function returns the smaller value out of two values.

    The two values must be Integers. If not
    you have to use an alternative function.
```

Show multiply again

Summary– Functions

- Functions are available as **built-in** to be used efficiently with any **Python objects**
- You can easily create your **own function** by using the **`def`** keyword and writing the block of statements that defines the **action** to perform
- Functions **return values** or "**`Nones`**"; return values are input to the program flow
- Functions allow the use of **arguments** and **default values** to improve their operability and their reusability

MultiplyFct: Fct in Fct (negativetest() in myMethodParKey()); PLUS student fct file; PLUS Linemaker as Fct