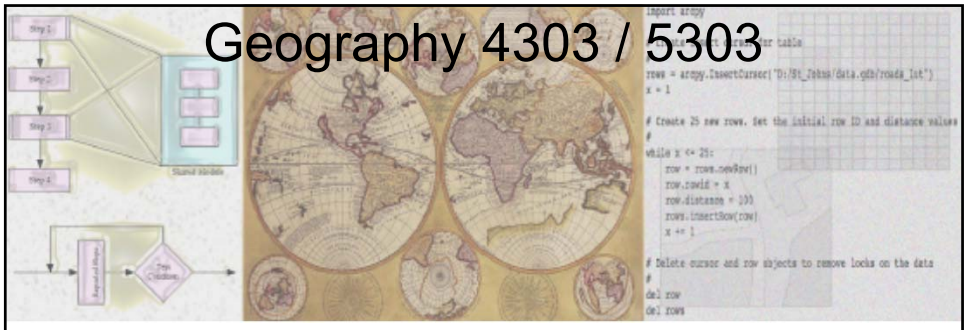


Geography 4303 / 5303

GIS PROGRAMMING FOR SPATIAL ANALYSIS

Class 05: Modules, Py Logic, Operators

Some Updates

- Project proposals
- Proposal presentations in two weeks
- Lab work on sampling – basic strategies…
- Pseudo codes

Last Lecture / Last Week

- We looked at **Functions**
- Modularity, abstraction, reusability and structure in your program
- Defining a function vs. calling the function
- Parameters and arguments, local/global variables; return values

Today ‘s Outline

- Create **user-specified modules**
- Fundamental rules of logical operators in general and their functionality
- Understanding **logical** and **arithmetic operators**
- How to make use of **logical** operators **to control flow** in complex **conditional constructs** and **iterations**

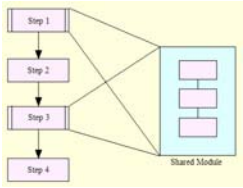
Learning Objectives

- How to create and use **modules**
- Shed light on the **logic** behind operators and the **operability** of logic elements
- **Use** these operators appropriately under constraints (precedence or **priority** rules)
- Get an important **foundation** for programming in general

Recall Again: 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

Modules



- **Reusing** code pieces (e.g., functions) in different programs
- "**Modularity** of Python": User-defined, external and from Python standard library
- Modules as upper level of organization
- **Access** points for programs to functionalities (variables or functions) contained in a module

Standard Library Modules

- Basically, a **module** is a file (.py) containing **functions** and/or **variables** (properties)
- These become available for use after loading the module (reading, executing code to make names available)
- Highest organization level
- Can be **imported** by another program (or module)
- **Standard library modules: e.g.,** `sys,os,string`

```
import math          #module math
math.sqrt(81)        #function sqrt of module
```

Conceptual idea mod/fct

How to import…

- `import` => the entire module is loaded (requires module prefix when calling methods)

```
>>> import math
>>> sqrt(81)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
NameError: name 'sqrt' is not defined
>>> math.sqrt(81)
9.0
```

- `from…import` => names of variables or functions to be loaded (prefixes not needed for calls)
- `from…import*`

```
>>> from math import*
>>> sqrt(81)
9.0
```

Example sys (System) 1

- Interaction with the Python system
- **`sys.path`:**

  Variable which lists all directories searched for a module after an **`import`** <module> statement
- File (module) found: Main block of the module is run (means: the module is **available** to us)

  first empty space = **current directory**

```
>>> sys.path
['', 'C:\\Programs\\ArcGIS\\bin', 'C:\\WINDOWS\\system32\\python24.zip',
'C:\\Programs\\Python24', 'C:\\Programs\\Python24\\DLLs', 'C:\\Programs\\Python24\\lib',
'C:\\Programs\\Python24\\lib\\plat-win', 'C:\\Programs\\Python24\\lib\\lib-tk',
'C:\\Programs\\Python24\\Lib\\site-packages\\pythonwin',
'C:\\Programs\\Python24\\lib\\site-packages',
'C:\\Programs\\Python24\\lib\\site-packages\\win32',
'C:\\Programs\\Python24\\lib\\site-packages\\win32\\lib',
'C:\\Programs\\Python24\\Lib\\site-packages']
```

User-specified sys Paths

- Store your custom modules in a directory
- Append this directory to the `sys.path` list before importing your module
- After Python session `sys.path` at default again

```
import sys
sys.path.append('C:\\Temp')
print sys.path
```

Modules and Program Flow

- Module that your are writing and running can **control the program flow**
- …represents the "**top-level"** module
- …can **import** other modules and use their functionality (**lower level** modules)
- …can use **built-in** functions and **standard lib** modules…

User-specified Modules (1)

- Create your **own module** and define its functions
- These functions become available for other programs (reuse) to extend functionality
- By using **`import`** you can work with them like with system modules if:

  (1) they are located in one of the paths listed under **`sys.path`**

  (2) they have the **extension *.py**

User-specified Modules (2)

- An **Example**:

  Create a *.py file, name it class10function.py and type:

```
def printTwice(n):
    print n, n
```

  Then save it under one of the sys.path directories or append a new path

```
>>> import class10function
>>> class10function.printTwice("Ahoj")
Ahoj Ahoj
```

- In the Interactive Window type

```
>>> import class10f printTwice
>>> class10function.
```

  Or extend sys.path by the workspace where it's located

Rules for Writing Code

- In your own modules write your **function definitions** at the **beginning of the file**
- "**Top-Level**" code: The code that calls your function
- …located at the **bottom** of the file OR in a **different module**
- This way the function definitions are read first (and the function name is known when you run the function)

Example levels (LineMaker as function)

Example `dir()` Function

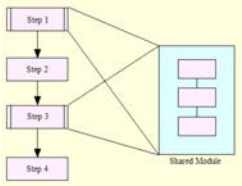
- The built-in function `dir()` is used to list the identifiers (function, classes, variables) that a **module** defines
- Returns a sorted **list of strings** of attributes and functions found in the module
- Current module if no argument supplied

```
>>> dir(class10function)
['__builtins__', '__doc__', '__file__', '__name__', 'printTwice']
```

Byte-Compiled .pyc Files

- Importing a module is costly
- Optimizations to create **byte-compiled files** with the extension **.pyc** with the same name as the module
- The next time importing it will be much **faster**. These byte-compiled files are **platform independent**

Summary– Modules



- So far, **functions** and **modules** (user-specified and built-in) are very important features in programming
- Modules can be **hierarchically** ordered by deciding where the "top-level" is
- Allow **reuse** of functions in other programs
- Allow **extending** and **specifying** functionality for **user-oriented** solutions
- **Structure**, **abstracting** and **readability**

Exercise class04 as modules

Programming Techniques

- Logic
- Conditional constructs & Decision-making
- Branching, redirecting program flow
- Repetitive operations (looping / iteration)
- Dialogs
- Error search

Very basic …

- **Statements** (logical lines) contain expressions
- These **expressions** consist of **operators** (functionality) and **operands** (data for the operator to work on)
- We had already a look at operands …

Arithmetic operators

- ‘+’, ‘-’, ‘*’, ‘**’, ‘/’
- Floor division: ‘//’ (Floor of the quotient)
- Modulo: ‘%’ (Remainder of a division)
- Shortcuts:

x = x + y	=> x += y
x = x - y	=> x -= y
x = x * y	=> x *= y
x = x / y	=> x /= y
x = x % y	=> x %= y

Some examples ...

Boolean / Logical Operators

- Boolean Expressions (or tests) turn into **true** (value 1) or **false** (value 0)
- They thus turn into Boolean outcomes
- Using **comparison operators**:

==	is equal (equal comparison, caution: not assignment!!)
!= or <>	is not equal
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to

```
>>> a = 3
>>> b = 4
>>> a <> b
True
>>> a > b
False
>>> a == b
False
```

Show why boolean...

Boolean / Logical Operators

- Three more logical operators: **and**, **or**, and **not**
- **Multiple conditions:** testing of variables to also create Boolean output
- Treated as **individual Boolean** expressions of which each (and) or at least one (or) has to turn into **true**

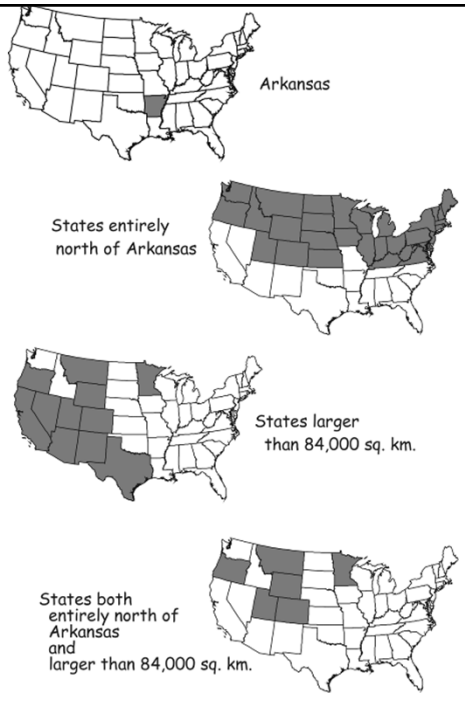
```
cVar > 2 and cVar < 100
```

- This expression is true if cVar is greater than 2 AND less than 100
- Conditional constructs or While loops
- Sets and set logic

```
>>> a = 3
>>> b = 5
>>> c = 92
>>> a == b or a < c
True
```

Venn principles; if example

Queries in a database

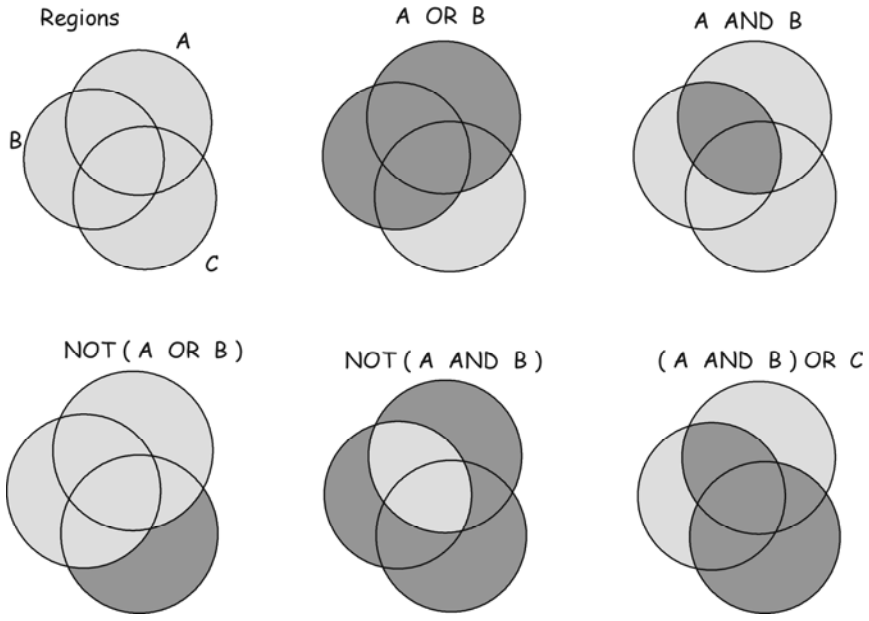


Set algebra:

<, >, =, <>

Boolean algebra:

disjunction (OR)
conjunction (AND)
complement/negation (NOT)



Overlaying sets (spatial and/or non-spatial ones)

Operator Precedence

- If combining operators in **complex** statements the **precedence** is important
- Lowest precedence: (**least binding**) on **top**
- Highest precedence: (**most binding**) **bottom**
- Parentheses …

Operator	Description
`lambda`	Lambda expression
`or`	Boolean OR
`and`	Boolean AND
`not` *x*	Boolean NOT
`in`, `not in`	Membership tests
`is`, `is not`	Identity tests
<, <=, >, >=, <>, !=, ==	Comparisons
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and subtraction
*, /, %	Multiplication, division, remainder
+*x*, -*x*	Positive, negative
~*x*	Bitwise not
**	Exponentiation
x.*attribute*	Attribute reference
x[*index*]	Subscription
x[*index*:*index*]	Slicing
f(*arguments*...)	Function call
(*expressions*...)	Binding or tuple display
[*expressions*...]	List display
{*key*:*datum*...}	Dictionary display
`'`*expressions*...`'`	String conversion

Example (+) and (<)

Order of Evaluation

- Decided by the **precedence** table (**default**)
- Order can be changed by using **parentheses**

  `a + b * c` `(a+b) * c`
- Operators of the same precedence in the same statement are evaluated from **left to the right**

An example…

Important Points 1

- Comparison operators are of lower **priority** than arithmetic operators: **`a + b < c * d`**
- Precedence when combining comparisons with **`and`**, **`or`** and **`not`**:
  - comparisons higher: **`a < c OR d < b`**
  - **`not`** highest, **`or`** lowest:

```
   a<b and not b==c or a>c
=> (a<b and (not b==c )) or a>c
```

Important Points 2

- "**Short circuit**" operators `and` and `or`:

  Evaluation is stopped as soon as outcome can be determined
- Evaluation from **left to right** in one logical line

  `x<1 and y>5 and z<8:`

  (z is not evaluated if y>5 is false)

```
>>> a = 2
>>> b = 6
>>> c = 13
>>> if a < b and a < c:
...     print "YES"
...
YES
>>> if a < b or a > c:
...     print "YES"
...
YES
```

Ho to find out:

a or b
Out[32]: 2

a and b
Out[33]: 6

Show first 2 examples in file ...

Queries using conditional constructs

- `slope=5`
- `elev=1600`
- `aspect=180`

```
if slope<10 and elev<2200 and aspect<200
if slope<10 and elev<1500 or aspect<200
```

Elements `in` / `not in`

- Comparison operators `in` and `not in` are used to check if a value is (not) in a sequence
- Boolean test?
- Where did you use this already?

```
>>> myList
[1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974,
1980, 1981, 1982, 1983, 1984, 1985, 1986]
>>> 1972 in myList
True
>>> 1945 in myList
False
```

Quick list example and then for loop in file

Object `is` / `is not`

- Operators `is` (==) and `is not` (!=) are used to check if two objects are equal
- Makes sense to check lists while (or better before) changing anything

Example List

```
>>> myList = range(1948, 1953)
>>> yourList = range(1947,1956)
>>> myList is yourList
False
>>> myList is not yourList
True
```

Example String

```
>>> r = "erst"
>>> s = "dann"
>>> r is s
False
>>>
```

Exists Tests

- `os.path.exists()` tests for existing paths returns Boolean output
- `os.path.isFile(fileName)` tests for files' existence
- `arcpy.Exists()` does so too (allows you to test for paths and files

```
if arcpy.Exists("RoadsBuff"):
   arcpy.Delete_management("RoadsBuff")
```

class05_testExists

Revisiting Flow Control

- Changing the **flow of execution**
- Different situations or testing outcomes determine **different directions** of programming flow
- Solving these things by "control flow statements" (**`if`, `for`** and **`while`**)

Decision Making 1

- Conditional statement: **if ... elif ... else**

```
if i == 3:
     print "i is 3"  #if block
elif i == 4:
     print "i is 4"  #optional elif block
else:
     print "i is not 3 and not 4"  #optional else block
```

- **Tests use logic operators (to work on operands) and result in Boolean outcomes**
- Nested if statements

Looping Using `While` 1

- Repeated execution of a block of statements as long as a condition is true

  Continues while a condition is true
- **Tests use logic operators (to work on operands) and result in Boolean outcomes**

  Example:

```
i = 1
while i < 10:
     print i
     i = i + 1  # i += 1


i = 1; j = 0
while i < 10 and j < 19:
     print i
     i = i + 1
     j = j + 2
```

class05_student_interactive

Counted Loop

- Iterate sequences and over each (“for each”) value in sequence
- Number of **iterations** = number of **list elements**
- Execution once for each value
- **Tests use logic operators (to work on operands) and result in Boolean outcomes**
- Nested loops

class05_forloopagain; overlay queries in list compreh.

`Break` Statement

- “break out of a loop”
- Execution of a loop is stopped even if loop condition is `true` or there are elements to be iterated over
- `Else` blocks are not executed

```
for i in range(0,4):
    print "Element", i ,"before BREAK"
    if i == 2:
        break

    print "Element", i, "after BREAK"
```

Class05_break

`Continue` Statement

- "skip the rest of the current loop block and continue to the next iteration of the same loop"

```
for i in range(0,4):
    print "Element", i ,"before CONTINUE"
    if i == 2:
        continue

    print "Element", i, "after CONTINUE"
```

Class05_continue

Summary

- While working with **operators** we have to pay attention to **precedence** and **logic**
- **Comparisons** can be complex and **combined** with each other
- These Logical constructs can be used for **process flow**, **conditional constructs** and **iteration** (while and counted looping)