GEOG 399L: Programming for Spatial Data Science

Lecture 2: Variables, data types, and structures

Email: jryan4@uoregon.edu
Office: 163A Condon Hall
Office hours: Monday 14:00-16:00

Variables, data types, and structures

Welcome to the **first demo** of the course! In these demos, the instructor will work through some key programming concepts and demonstrate with plenty of examples. Understanding these concepts will be required to complete the weekly assignments.

Variables, data types, and structures

Welcome to the **first demo** of the course! In these demos, the instructor will work through some key programming concepts and demonstrate with plenty of examples. Understanding these concepts will be required to complete the weekly assignments.

It's a calculator

A Python interpreter can be used as a calculator.

```
3 + 3
```

```
6
```

Variables, data types, and structures

Welcome to the **first demo** of the course! In these demos, the instructor will work through some key programming concepts and demonstrate with plenty of examples. Understanding these concepts will be required to complete the weekly assignments.

It's a calculator

A Python interpreter can be used as a calculator.

```
3 + 3
```

```
6
```

```
3 / 3
```

```
1.0
```

Variables

This is great but to do something more useful with data, we need to assign its value to a **variable**. Variables are one of the fundamental building blocks of Python. A variable is like a tiny container where you store values and data, such as filenames, words, numbers, collections of words and numbers. In Python, we can assign a variable using the equals sign `=`. For example, we can track the height of a tree by assigning an integer value of `30` to a variable `height_m`.

Variables

This is great but to do something more useful with data, we need to assign its value to a **variable**. Variables are one of the fundamental building blocks of Python. A variable is like a tiny container where you store values and data, such as filenames, words, numbers, collections of words and numbers. In Python, we can assign a variable using the equals sign `=`. For example, we can track the height of a tree by assigning an integer value of `30` to a variable `height_m`.

```
height_m = 30
```

Variables

This is great but to do something more useful with data, we need to assign its value to a **variable**. Variables are one of the fundamental building blocks of Python. A variable is like a tiny container where you store values and data, such as filenames, words, numbers, collections of words and numbers. In Python, we can assign a variable using the equals sign `=`. For example, we can track the height of a tree by assigning an integer value of `30` to a variable `height_m`.

```
height_m = 30
```

We can also make a **string** variable by adding single (') or double quotes (") around some text.

```
tree = 'douglas_fir'
```

Variables

This is great but to do something more useful with data, we need to assign its value to a **variable**. Variables are one of the fundamental building blocks of Python. A variable is like a tiny container where you store values and data, such as filenames, words, numbers, collections of words and numbers. In Python, we can assign a variable using the equals sign `=`. For example, we can track the height of a tree by assigning an integer value of `30` to a variable `height_m`.

```
height_m = 30
```

We can also make a **string** variable by adding single (') or double quotes (") around some text.

```
tree = 'douglas_fir'
```

Note

Python is “dynamically typed” meaning that it automatically interprets the correct data type at run-time

Variable usage

Once we have data stored with variable names, we can make use of it in calculations. We may want to store our tree height value in feet as well as meters

```
height_ft = height_m * 3.281
```

Variable usage

Once we have data stored with variable names, we can make use of it in calculations. We may want to store our tree height value in feet as well as meters

```
height_ft = height_m * 3.281
```

Likewise, we might want to add a suffix to our tree so we can identify it later.

```
tree1 = tree + '_1'
```

Variable names

It is good practice to make variable names as descriptive as possible. They can include:

✅ upper and lower-case letters (`a-z`, `A-Z`)

✅ digits (`0-9`)

✅ underscores (`_`)

However, variable names **cannot** include:

❌ other punctuation (`-.!?@`)

❌ spaces ( )

❌ a reserved Python word (e.g. `print`, `type`)

Built-in functions

Python has a number of built-in functions to carry out common tasks with data and variables. For example, we can use the `print` function to **display information** to the screen.

Built-in functions

Python has a number of built-in functions to carry out common tasks with data and variables. For example, we can use the `print` function to **display information** to the screen.

```
height_ft = height_m * 3.281
```

```
print(height_ft)
```

```
98.43
```

Built-in functions

Python has a number of built-in functions to carry out common tasks with data and variables. For example, we can use the `print` function to **display information** to the screen.

```
height_ft = height_m * 3.281
```

```
print(height_ft)
```

```
98.43
```

```
tree1 = tree + '_1'
```

```
print(tree1)
```

```
douglas_fir_1
```

Built-in functions

Python also has a built-in function called `type` which outputs a value's **data type**

```
type(height_ft)
```

```
float
```

```
type(tree1)
```

```
str
```

> **Note**
>
> Note the use of parentheses `()` to let the function know which value we want to display.

Built-in functions

The three most common data types that we will come across in Spatial Data Science are integer numbers (`int`), floating-point numbers (`float`) and strings (`str`). We may also encounter boolean data types (`bool`) which may have one of two values, `True` or `False`.

Built-in functions

The three most common data types that we will come across in Spatial Data Science are integer numbers (`int`), floating-point numbers (`float`) and strings (`str`). We may also encounter boolean data types (`bool`) which may have one of two values, `True` or `False`.

Data Type	Explanation	Example
String	Text	`"july"`
Integer	Whole numbers	`42`
Float	Decimal numbers	`84.2`
Boolean	True/False	`False`

Built-in functions

We can use another built-in function to change the data type of our variable (e.g. to `int`).

```
int(height_ft)
```

```
print(height_ft)
```

```
98.43
```

Built-in functions

We can use another built-in function to change the data type of our variable (e.g. to `int`).

```
int(height_ft)
```

```
98
```

```
print(height_ft)
```

```
98.43
```

Built-in functions

We can use another built-in function to change the data type of our variable (e.g. to `int`).

```
evergreen = True
type(evergreen)
```

```
bool
```

```
int(evergreen)
```

Built-in functions

We can use another built-in function to change the data type of our variable (e.g. to `int`).

```
evergreen = True
type(evergreen)
```

```
bool
```

```
int(evergreen)
```

```
1
```

Built-in functions

Function	Description
abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
callable()	Returns True if the specified object is callable, otherwise False
chr()	Returns a character from the specified Unicode code.
classmethod()	Converts a method into a class method
compile()	Returns the specified source as an object, ready to be executed
complex()	Returns a complex number

Take a breather

Lists

Sometimes we might want to store a **collection of items** in a single variable. The simplest type of collection in Python is a `list` which can be defined using square brackets `[` and commas `,`.

Lists

Sometimes we might want to store a **collection of items** in a single variable. The simplest type of collection in Python is a `list` which can be defined using square brackets `[` and commas `,`.

```
places = ['Eugene', 'Veneta', 'Noti', 'Mapleton', 'Florence']
```

```
type(places)
```

```
list
```

Indexing

We can access individual items in a list using an **index** value. An index value is a **number** that refers to a position in the list. We can access the **second** value of the list by running:

```
places[1]
```

```
places = ['Eugene', 'Veneta', 'Noti', 'Mapleton', 'Florence']
```

Indexing

We can access individual items in a list using an **index** value. An index value is a **number** that refers to a position in the list. We can access the **second** value of the list by running:

```
places[1]
```

```
'Veneta'
```

```
places = ['Eugene', 'Veneta', 'Noti', 'Mapleton', 'Florence']
```

> **Note**
>
> Python uses **zero-based indexing** meaning that the first item of a list is accessed using the index value **0**.

Indexing

We can find the number of items in a list using the `len()` function.

```
len(places)
```

```
5
```

```
places = ['Eugene', 'Veneta', 'Noti', 'Mapleton', 'Florence']
```

Indexing

We can find the number of items in a list using the `len()` function.

```
len(places)
```

```
5
```

If we know the number of items in the list, we can access the last item by running:

```
places[4]
```

```
places = ['Eugene', 'Veneta', 'Noti', 'Mapleton', 'Florence']
```

Indexing

We can find the number of items in a list using the `len()` function.

```
len(places)
```

```
5
```

If we know the number of items in the list, we can access the last item by running:

```
places[4]
```

```
'Florence'
```

```
places = ['Eugene', 'Veneta', 'Noti', 'Mapleton', 'Florence']
```

Slicing

Slicing is similar to indexing except that we are looking to return a subset of items based their indices. We use a colon `:` to slice lists. For example, we can return the second and third items from out list.

```
places[1:3]
```

```
places = ['Eugene', 'Veneta', 'Noti', 'Mapleton', 'Florence']
```

Slicing

Slicing is similar to indexing except that we are looking to return a subset of items based their indices. We use a colon `:` to slice lists. For example, we can return the second and third items from out list.

```
places[1:3]
```

```
['Veneta', 'Noti']
```

> **Note**
>
> The first number represents index item to include and second number is the index item to stop at *without including it in the slice*.

```
places = ['Eugene', 'Veneta', 'Noti', 'Mapleton', 'Florence']
```

Slicing

Leaving either side of the colon blank means start from (or go to) the end of the list. For example:

```
places[3:]
```

```
['Mapleton', 'Florence']
```

```
places = ['Eugene', 'Veneta', 'Noti', 'Mapleton', 'Florence']
```

Slicing

Leaving either side of the colon blank means start from (or go to) the end of the list. For example:

```
places[3:]
```

```
['Mapleton', 'Florence']
```

```
places[:3]
```

```
['Eugene', 'Veneta', 'Noti']
```

```
places = ['Eugene', 'Veneta', 'Noti', 'Mapleton', 'Florence']
```

Stepping

We can use double colons `::` to set the interval at which items are included in the slice. So we can get every **second** item in the list (starting from index `0` by running:

```
places[::2]
```

```
['Eugene', 'Noti', 'Florence']
```

```
places = ['Eugene', 'Veneta', 'Noti', 'Mapleton', 'Florence']
```

Stepping

We can use double colons `::` to set the interval at which items are included in the slice. So we can get every **second** item in the list (starting from index `0` by running:

```
places[::2]
```

```
['Eugene', 'Noti', 'Florence']
```

If we wanted to start at index `1`, we could run:

```
places[1::2]
```

```
['Veneta', 'Mapleton']
```

```
places = ['Eugene', 'Veneta', 'Noti', 'Mapleton', 'Florence']
```

List methods

Methods are similar to the built-in functions we used earlier but are called on the object itself.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

List methods

We can add items to a list using the `.append` method.

```
places.append("Coos Bay")
```

```
print(places)
```

```
['Eugene', 'Veneta', 'Noti', 'Mapleton', 'Florence', 'Coos Bay']
```

List methods

Other useful list methods

We can use methods to count the number of occurences of an item:

```
places.count('Eugene')
```

```
1
```

List methods

Other useful list methods

We can use methods to count the number of occurences of an item:

```
places.count('Eugene')
```

```
1
```

Find the index value of a given item in a list:

```
places.index('Eugene')
```

```
0
```

List methods

Reverse list

```
places.reverse()
print(places)
```

```
['Coos Bay', 'Veneta', 'Noti', 'Mapleton', 'Florence', 'Eugene']
```

Sort a list alphabetically

```
places.sort()
print(places)
```

```
['Coos Bay', 'Eugene', 'Florence', 'Mapleton', 'Noti', 'Veneta']
```

Remove an item from the list

```
places.remove('Coos Bay')
print(places)
```

```
['Eugene', 'Florence', 'Mapleton', 'Noti', 'Veneta']
```

Take a breather

Formatting notebooks

The second demo of this week provides an introduction to formatting assignment submissions. Jupyter Notebooks are powerful because we can add documentation in **Markdown**, a lightweight markup language for creating formatted text using a plain-text editor.

Cells

There are two main types of cell in Jupyter Notebooks: Code and Markdown.

Formatting notebooks

The second demo of this week provides an introduction to formatting assignment submissions. Jupyter Notebooks are powerful because we can add documentation in **Markdown**, a lightweight markup language for creating formatted text using a plain-text editor.

Cells

There are two main types of cell in Jupyter Notebooks: Code and Markdown.

In **code** cells, the text is treated as statements in a programming language of current kernel (Python in our case). When such cell is run, the result is displayed in an output cell.

Formatting notebooks

The second demo of this week provides an introduction to formatting assignment submissions. Jupyter Notebooks are powerful because we can add documentation in **Markdown**, a lightweight markup language for creating formatted text using a plain-text editor.

Cells

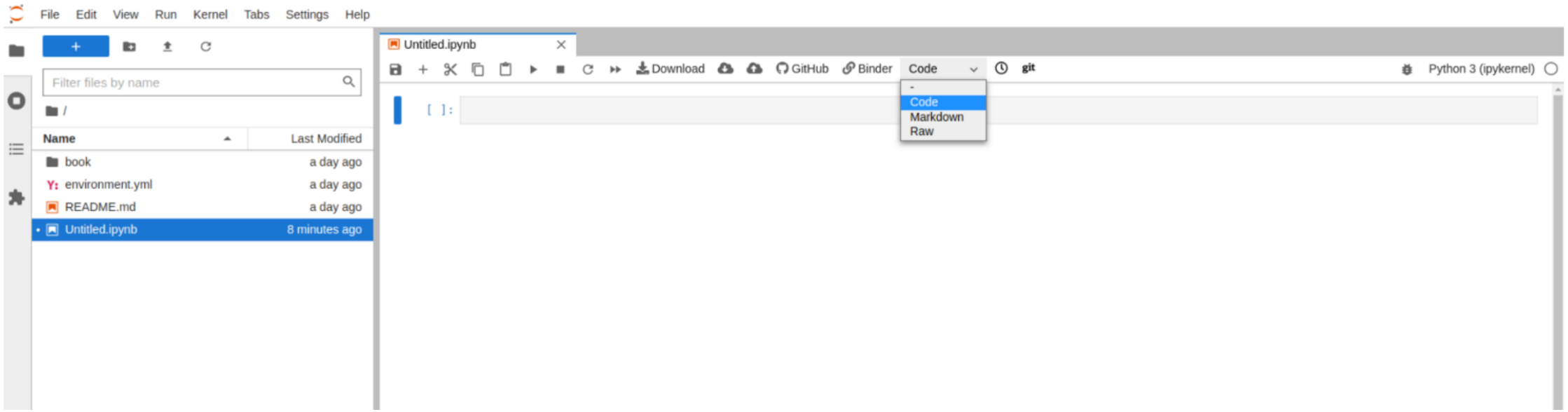
There are two main types of cell in Jupyter Notebooks: Code and Markdown.

In **code** cells, the text is treated as statements in a programming language of current kernel (Python in our case). When such cell is run, the result is displayed in an output cell.

In **Markdown** cells, the text is formatted using markdown language. All kinds of formatting features are available like making text **bold** and *italic*, displaying ordered or unordered list, rendering tabular contents etc. Markdown cells are especially useful for providing documentation to a notebook.

Formatting notebooks

We can choose the cell type by clicking on the drop-down list at the the top of the page.



Formatting notebooks

Markdown

Headings

We can add headings to our notebooks using the number sign (#) followed by a blank space:

```
# for titles

## for major headings

### for subheadings

#### for 4th level subheadings
```

Formatting notebooks

Emphasis

We can format text as bold, italic, or monospace font using underscores, asterisks, and grave accents:

```
Bold text: __string__ or **string**

Italic text: _string_ or *string*

Monospace text: `string`
```

Formatting notebooks

Lists

We can make bullet point lists using hyphens or asterisks followeed by a space. Each bullet point must be on its own line.

```
- A hyphen (-)
* An asterisk (*)
```

Formatting notebooks

Graphics

We can add graphics, figures, and images to our notebook using the following syntax:

```
![Image title](filename.png)
```

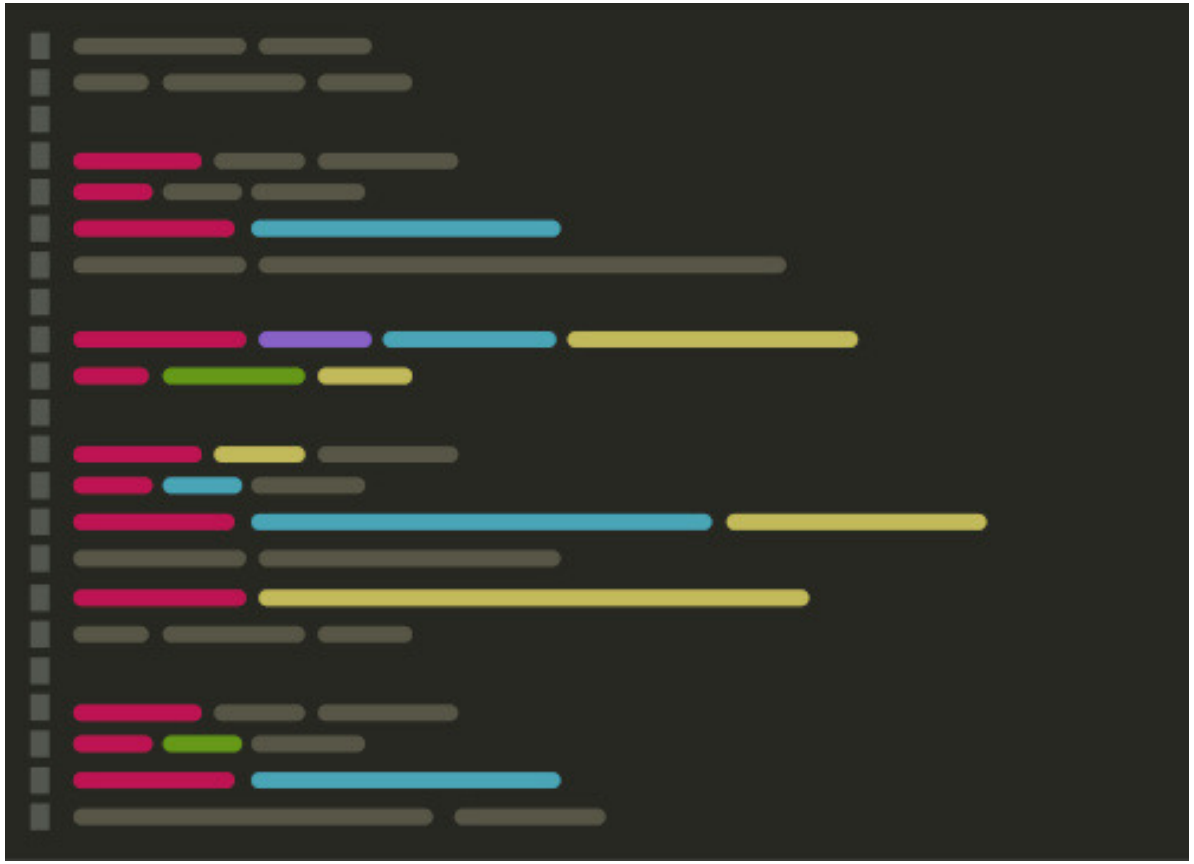
Formatting notebooks

Tables

We can even make tables:

```
| This | is    |
|------|-------|
| a    | table |
```

Next time: NumPy arrays



Email: jryan4@uoregon.edu
Office: 163A Condon Hall
Office hours: Monday 14:00-16:00