Lecture 1: Introduction, Variables, Data Types

Finlay Maguire (finlay.maguire@dal.ca)

TAs: Ehsan Baratnezhad (ethan.b@dal.ca); Precious Osadebamwen (precious.osadebamwen@dal.ca)

Overview

- What are the goals of this course?
- What is scientific programming?
- Why are we learning python?
- How is the course going to be structured and assessed?
- Expressions
- Variable assignment
- Data/value types
- Casting types
- Basic functions

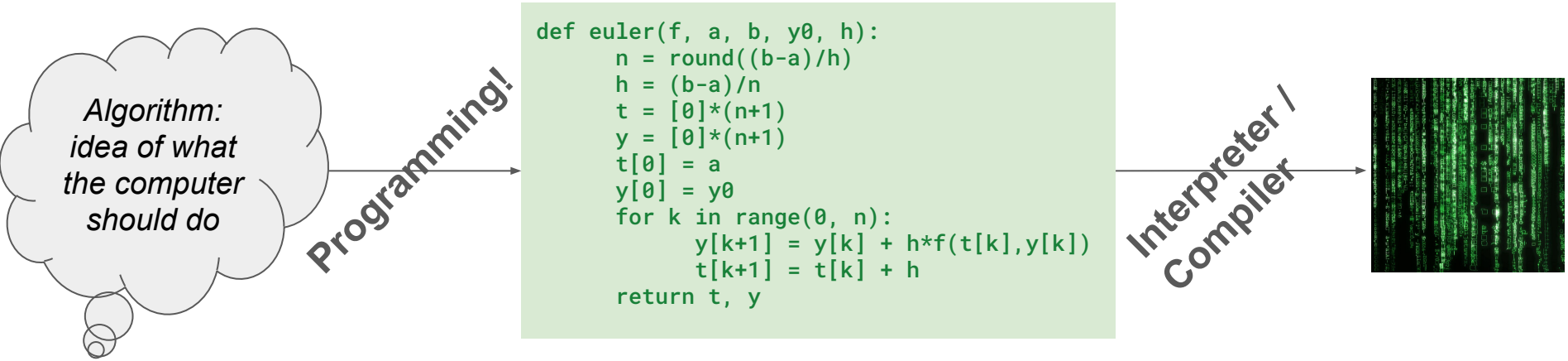
~~Computer Modelling for Scientists~~ Introduction to Programming with Applications for Scientists

The aim of this course is to prepare students to represent **scientific questions** as **computational problems** and apply **python-based** programming solutions. Specifically:

1. Read, test, and debug small to medium-size python programs.
2. Plan and develop computational solutions to practical scientific problems.
3. Perform basic data processing and visualization using widely-used python libraries
4. Apply basic ideas of computational complexity and optimisation to create more efficient programs.
5. Understand best practices for performing reproducible computational analyses with high quality code.

What is programming?

Programming is telling a computer exactly what to do.



I want to solve this ODE using Euler's method

I write some python code that tells the computer how to do this

*Computer does **EXACTLY** what I told it*

Programming languages are notation for computational processes

```
def euler(f, a, b, y0, h):
        n = round((b-a)/h)
        h = (b-a)/n
        t = [0]*(n+1)
        y = [0]*(n+1)
        t[0] = a
        y[0] = y0
        for k in range(0, n):
                y[k+1] = y[k] + h*f(t[k],y[k])
                t[k+1] = t[k] + h
        return t, y
```

- Similar: Calculus as a notation system for a certain types of mathematics.
- Alternative notations/languages can be used to achieve the same thing

Syntax: rules that define how notation is combined to generate valid code in a specific language

	Newton	Leibniz
Differentiation	$\dot{y} = \frac{dy}{dt}$	$\frac{d(f(x))}{dx}$ or $\frac{dy}{dx}$
Integration	$\overline{x}$ or $\boxed{x}$	$\int_a^b f(x)\,dx$

https://www.cs.cmu.edu/~ckingsf/class/02201-f15/lectnotes/lec0-intro.pdf

Programming helps you think about processes

- Most programming involve the same concepts:
  - Making choices:
    - Do A if B otherwise to C.
    - (Go to the store if it's raining, otherwise go running.)
  - Repeating things: Do A 100 times.
    - Do A until B.
    - (Run around the track until you are tired.)
  - Defining new operations in terms of existing ones:
    - When I say "Do A", you should do B,C,D,E.
    - (When I say "Go to the store", you should "drive there, and buy cookies, milk, cake, and chocolate.")

This course will involve you learning how to translate scientific problems into these type of concepts.

https://www.cs.cmu.edu/~ckingsf/class/02201-f15/lectnotes/lec0-intro.pdf

Why is programming relevant to science?

All modern science is a bit computational!

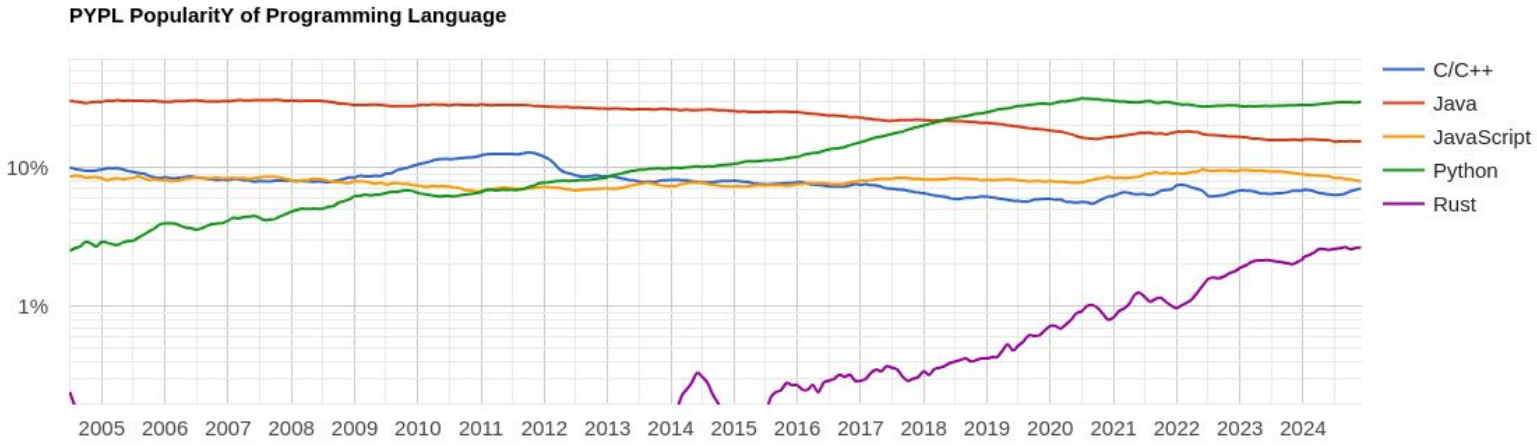
“Science rests on data, processing data needs software.”

- Computation behind many breakthroughs e.g., Higgs Boson, Human Genome, Structural Prediction, Climate Change, Deep Learning, all of modern statistics!

- Consider a typical scientific workflow:
  - Collect data (experiments/observations etc)
  - Process the data (fit to a model, extract parameters, etc)
  - Compare the data to your hypothesis (simulations, statistical analysis)
  - Visualise the results (make plots, generate summaries)
  - Write paper/report

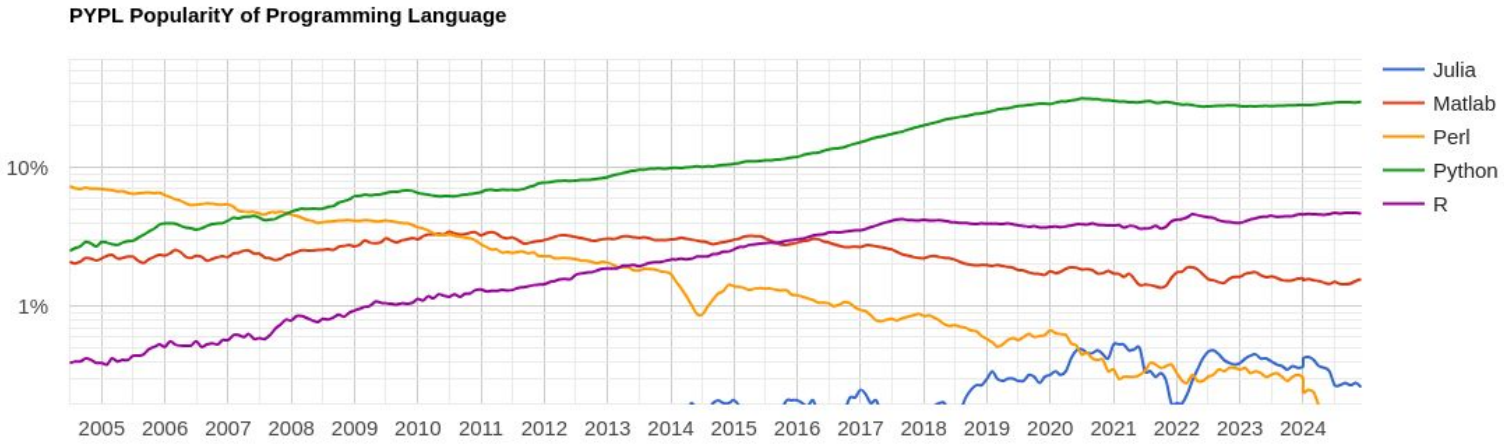
Why are we learning python?

Python is a very widely-used programming language



https://pypl.github.io/PYPL.html

Python is the most popular language in science



https://pypl.github.io/PYPL.html

Popularity = lots of other free and open code you can use

PyPI Stats

Search

All packages
Top packages

Track packages

Most downloaded PyPI packages

Project name	Sum of release files (bytes)
All of PyPI	24.5 TB

Most downloaded past **day**.

1	boto3	39,308,548
2	urllib3	17,847,737
3	botocore	16,457,667
4	requests	14,880,665
5	setuptools	13,576,925
6	certifi	13,523,318
7	charset-normalizer	13,348,032
8	grpcio-status	13,193,013
9	aiobotocore	12,753,682
10	python-dateutil	12,745,612
11	typing-extensions	12,643,496
12	idna	12,600,113
13	packaging	11,686,260
14	s3transfer	11,560,158
15	s3fs	10,540,997
16	fsspec	10,358,309
17	six	10,268,421
18	numpy	9,984,889
19	pip	9,684,664
20	pyyaml	8,807,869

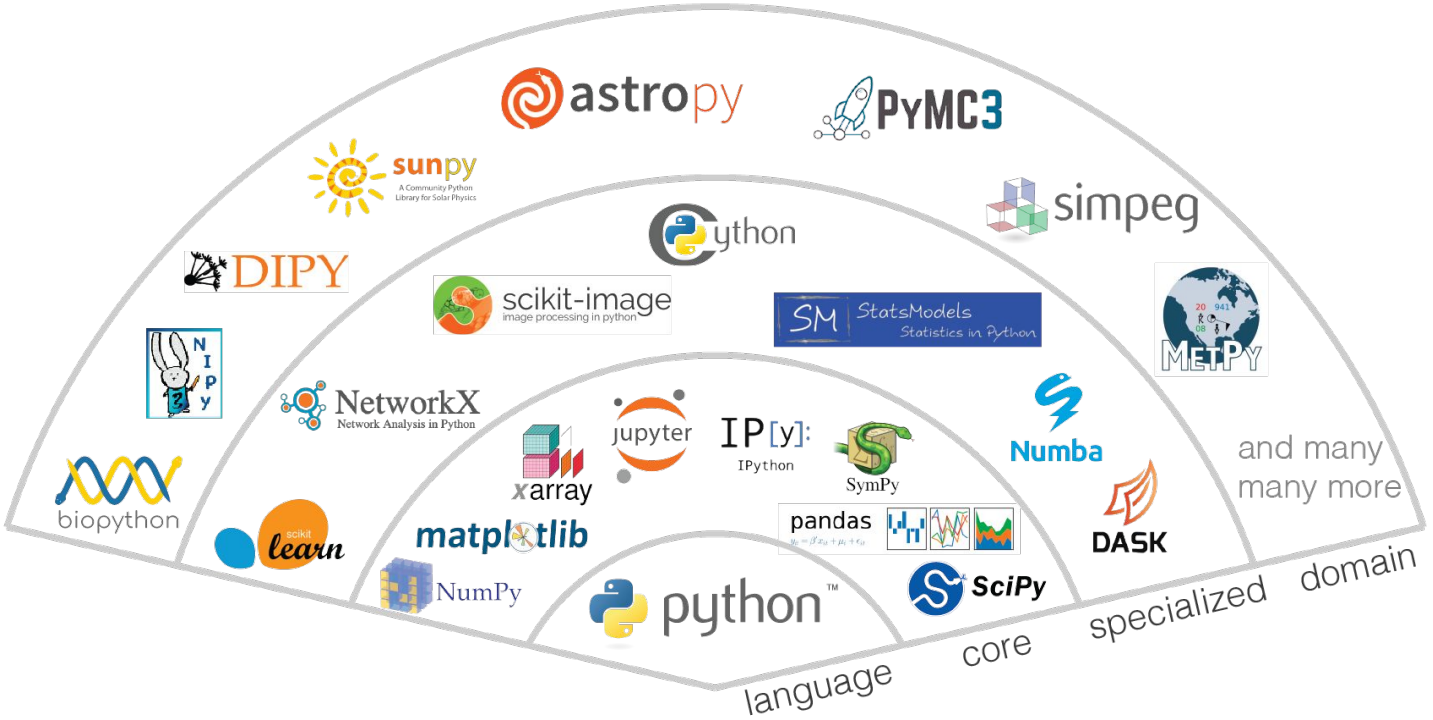
Most downloaded past **week**.

1	boto3	283,650,436
2	urllib3	126,076,945
3	botocore	118,252,758
4	requests	111,021,589
5	setuptools	98,961,104
6	certifi	98,449,806
7	charset-normalizer	94,427,364
8	python-dateutil	93,335,047
9	typing-extensions	91,985,630
10	idna	91,419,007
11	aiobotocore	90,100,029
12	grpcio-status	89,632,054
13	packaging	86,350,367
14	s3transfer	81,693,250
15	numpy	78,165,048
16	s3fs	74,625,329
17	fsspec	73,866,928
18	six	73,529,747
19	pip	64,659,900
20	wheel	63,818,242

Most downloaded past **month**.

1	boto3	1,434,433,895
2	urllib3	635,710,067
3	botocore	607,523,707
4	requests	576,088,457
5	setuptools	522,445,387
6	certifi	512,837,115
7	charset-normalizer	493,201,918
8	idna	482,891,737
9	packaging	474,755,490
10	typing-extensions	472,700,688
11	python-dateutil	472,610,545
12	aiobotocore	424,125,379
13	grpcio-status	416,964,863
14	numpy	398,480,451
15	s3transfer	394,751,243
16	six	381,133,793
17	fsspec	348,843,634
18	s3fs	344,216,611
19	pyyaml	339,878,190
20	wheel	324,959,038

Big ecosystem of science/data-focused code available



https://jupytearth.org/jupyter-resources/introduction/ecosystem.html

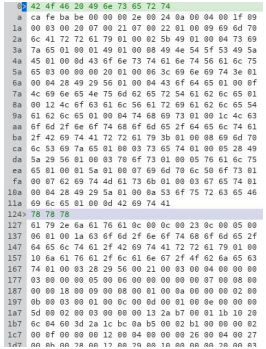
Python lets you do a lot with a little code

Low-Level Languages	Medium-Level Languages	High-Level Languages
Binary Code	C, C++	Cobol, Python, Pascal & JAVA
Machine Language	Assembly Language	

```python
def isPrime(n):
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

def prime(n: int) -> int:
    if n == 1:
        return 2
    p = 3
    pn = 1
    while pn < n:
        if isPrime(p):
            pn += 1
        p += 2
    return p-2

if __name__ == '__main__':
    print(prime(10001))
```



https://unstop.com/blog/what-is-programming-language

Rosettacode

Python has relatively concise and clear syntax

“Hello World”

- In Java, C, C++ a lot of “{“, “}” and “;” are needed
- Java tends to have a lot of “public...” details that need to be spelled out
- Python is concise

C

```c
#include <stdio.h>

int main(int argc, char **argv) {
    printf("Hello World");
    return 0;
}
```

C++

```cpp
#include <iostream>
using namespace std;

int main(int argc, char** argv) {
    cout << "Hello, World!";
    return 0;
}
```

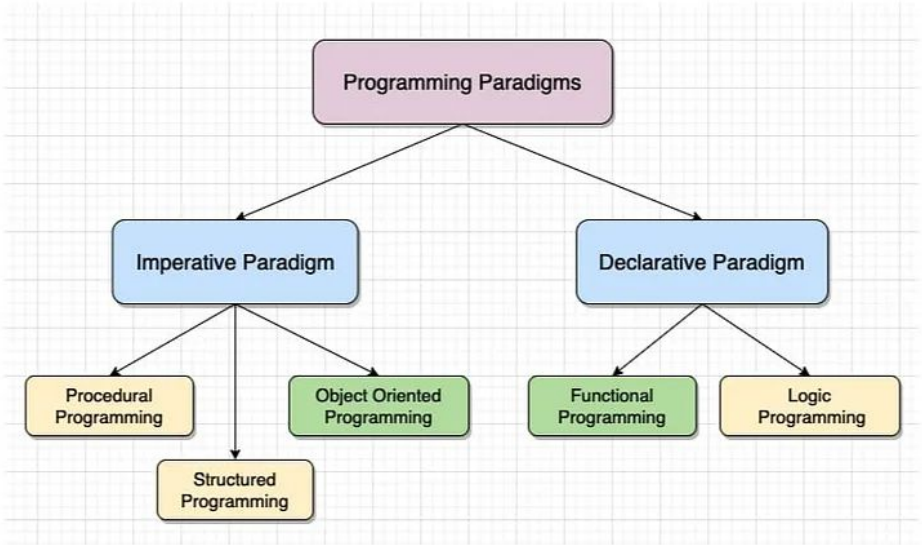
Java

```java
public class HelloWorld {
    public static void main( String[] args ) {
        System.out.println( "Hello World!" );
        System.exit( 0 );
    }
}
```

Python 3

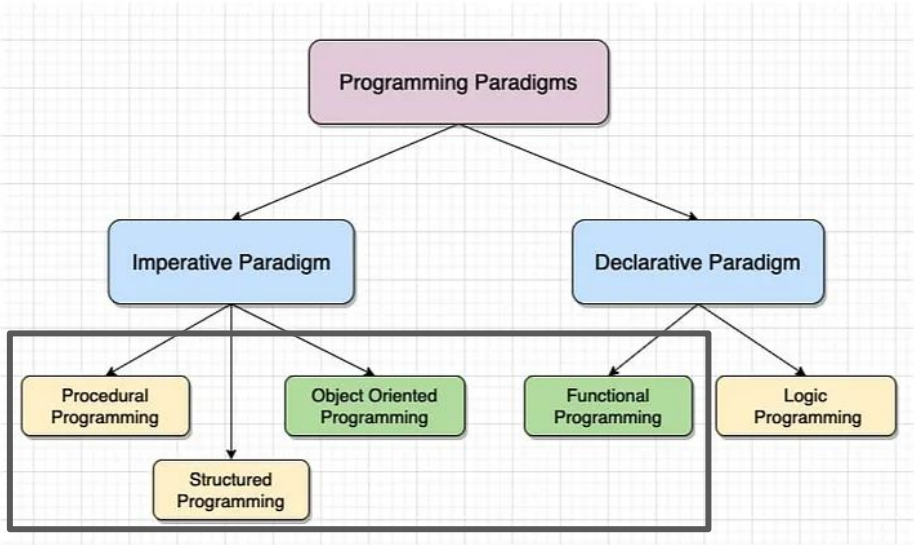
```python
print("Hello world")
```

Python is a flexible language



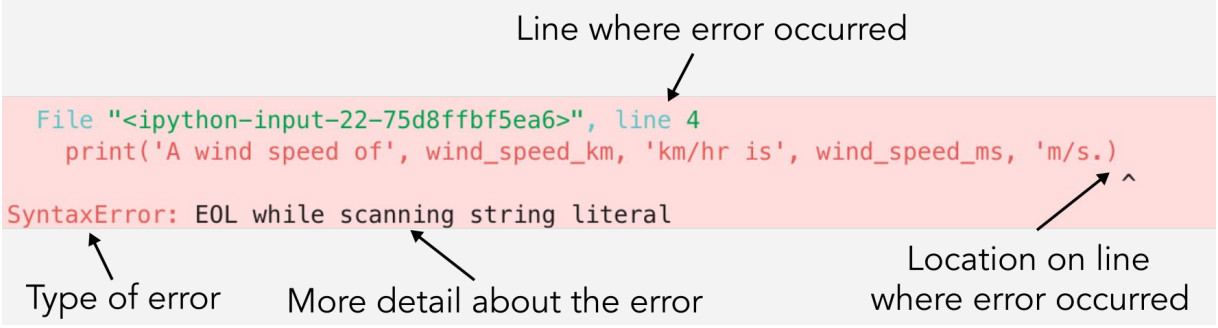
https://medium.com/@Ariobarxan/what-is-a-programming-paradigm-ec6c5879952b

Python is a flexible language



https://medium.com/@Ariobarxan/what-is-a-programming-paradigm-ec6c5879952b

Python has relatively simple error messages!



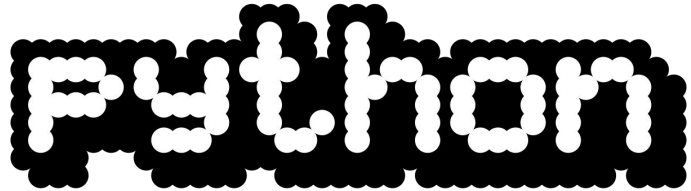
VS

```
<source>:27:20: error: no member named 'getName' in 'Foo'
                return object.getName();
                       ~~~~~~ ^
<source>:25:8: note: in instantiation of member function 'Object::Model<Foo>::getName' requested here
       Model(const T& t) : object(t) {}
       ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/ext/new_allocator.h:136:23: note: in instantiation of
member function 'Object::Model<Foo>::Model' requested here
        { ::new((void *)__p) _Up(std::forward<_Args>(__args)...); }
                      ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/alloc_traits.h:475:8: note: in instantiation of
function template specialization '__gnu_cxx::new_allocator<Object::Model<Foo> >::construct<Object::Model<Foo>, const Foo>' requested here
       { __a.construct(__p, std::forward<_Args>(__args)...); }
         ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/shared_ptr_base.h:526:30: note: in instantiation of
function template specialization 'std::allocator_traits<std::allocator<Object::Model<Foo> > >::construct<Object::Model<Foo>, const Foo>' requested here
         allocator_traits<_Alloc>::construct(__a, _M_ptr(),
                                  ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/shared_ptr_base.h:637:18: note: in instantiation of
function template specialization 'std::_Sp_counted_ptr_inplace<Object::Model<Foo>, std::allocator<Object::Model<Foo> >,
__gnu_cxx::_S_atomic>::_Sp_counted_ptr_inplace<const Foo>' requested here
         ::new (__mem) _Sp_cp_type(std::move(__a),
                  ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/shared_ptr_base.h:1294:14: note: (skipping 1 context
in backtrace; use -ftemplate-backtrace-limit=0 to see all)
       : _M_ptr(), _M_refcount(__tag, (_Tp*)0, __a,
                ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/shared_ptr.h:344:4: note: in instantiation of
function template specialization 'std::__shared_ptr<Object::Model<Foo>, __gnu_cxx::_S_atomic>::__shared_ptr<std::allocator<Object::Model<Foo> >, const
Foo>' requested here
       : __shared_ptr<_Tp>(__tag, __a, std::forward<_Args>(__args)...)
         ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/shared_ptr.h:690:14: note: in instantiation of
function template specialization 'std::shared_ptr<Object::Model<Foo> >::shared_ptr<std::allocator<Object::Model<Foo> >, const Foo>' requested here
       return shared_ptr<_Tp>(_Sp_make_shared_tag(), __a,
              ^
/opt/compiler-explorer/gcc-7.2.0/lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/shared_ptr.h:706:19: note: in instantiation of
function template specialization 'std::allocate_shared<Object::Model<Foo>, std::allocator<Object::Model<Foo> >, const Foo>' requested here
```

Using Python 3 in this course

Created by Guido van Rossum

- Python 1 (1991-2000)
- Python 2 (2000-2010) v2.7:
  - New features led to explosion in popularity
  - Move to more community-oriented development
- Python 3 (2008-):
  - Non-backward compatible clean-up of language with major changes
  - Currently v3.12.3
  - No plans for **python 4!**



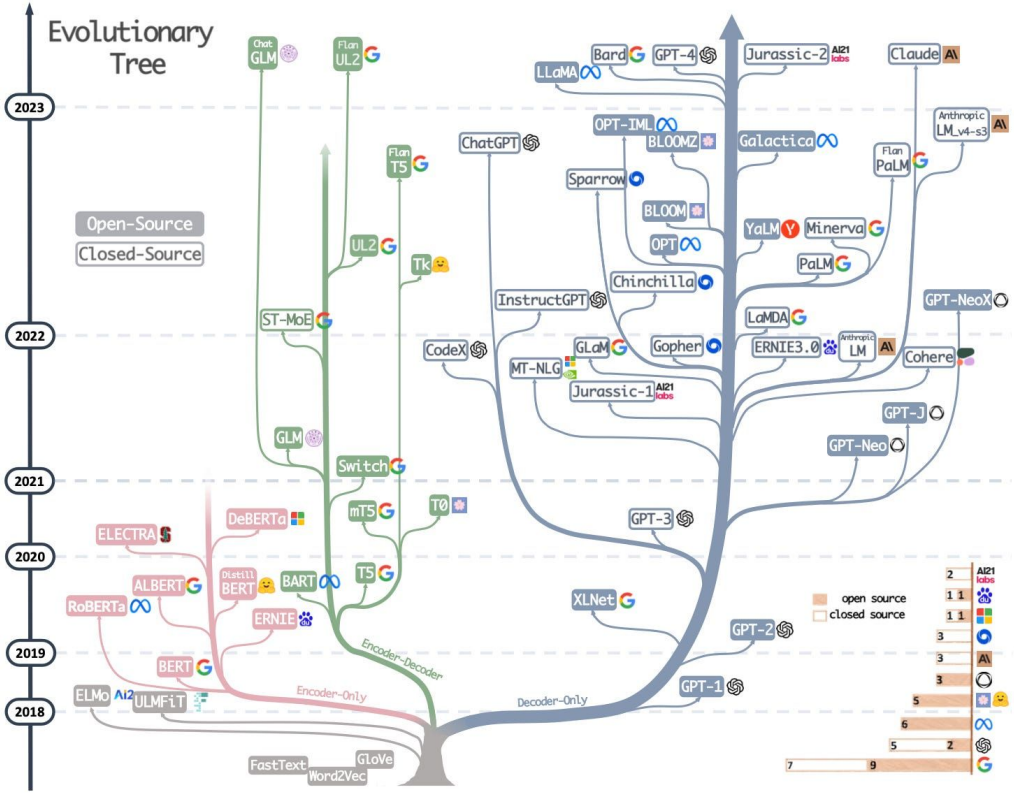
python™

Python 2	Python 3
print 42	print(42)
int = C long (32 bits)	int = arbitrary number of digits (= named “long” in Python 2)
7/3 → 2 returns “int”	7/3 → 2.333... returns “float”
range() returns list (memory intensive)	range() returns iterator (memory efficient; xrange in Python 2)

https://gsbrodal.github.io/ipsa/slides/all-slides.pdf
Wikimedia python logos

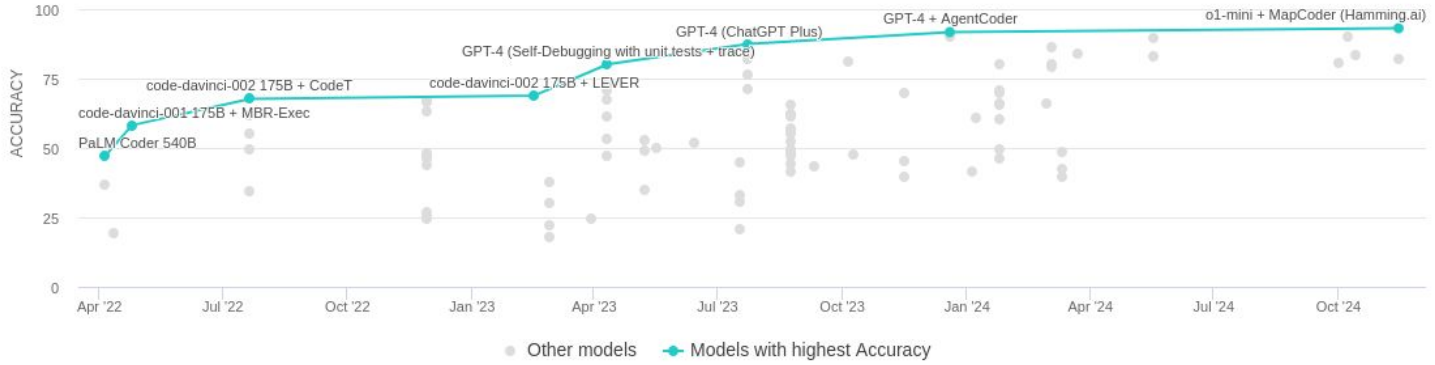
Learning python in 2025

The Large Language Model(s) in the Room



Yang, Jingfeng, et al. "Harnessing the power of llms in practice: A survey on chatgpt and beyond." ACM Transactions on Knowledge Discovery from Data 18.6 (2024): 1-32.

LLMs are quite good at basic python but you (probably) aren’t!



https://paperswithcode.com/sota/code-generation-on-mbpp

So, why bother learning to program at all?

LLMs have limitations:

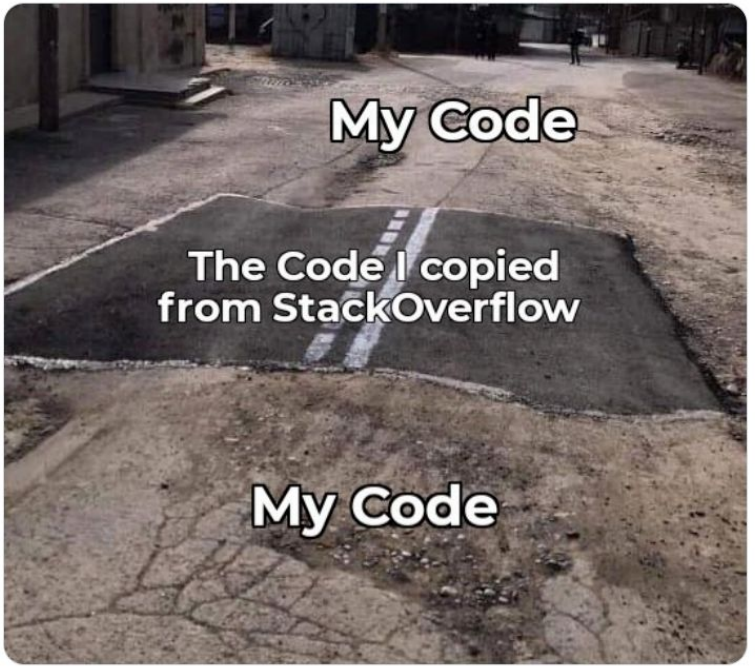
- Performance worse on more complicated tasks
- Performance worse on new languages/libraries/problem
- Hallucinations and combining code mean still need to debug
- Cross-domain challenging (e.g., scientific expertise => programming problem)
- **Plagiarism is still plagiarism…**

So:

- You are still responsible for your code (in university/research/work)
- You need to be able to understand and debug code
- Writing (not just) code helps you better understand a problem
- Doing is a very effective way of learning

Problem is not new: Stackoverflow Programmer

https://img.ifunny.co/images/b419ffea22ab2b2fee0b65747f7d9eebd16fa83841c2bdc9e26e16321b92eb82_1.jpg



https://programmerhumor.io/wp-content/uploads/2023/07/programmerhumor-io-stackoverflow-memes-programming-memes-630edb2fd22b51c.png

How is this course going to work?

CSCI2202 attempts a realistic compromise

- Lab practicals (**50%**) to get you hands-on experience. You can use whatever resources you want for lab practical assignments with **2 conditions**:

  1. **IF** you do so in a way that genuinely helps you learn

  2. **IFF** you cite your sources (including prompts and version) using inline comments

<u>Code you can't explain with no attribution is an academic integrity offence</u>

- Mid-term/Final (**50%**) are **in-person, on paper,** and **closed book**:
  - Focusing on evaluating your understanding of basic python code
  - Ability to design a program to solve a scientific problem
  - **NOT:** making you write code on paper

CSCI2202 Course Structure

Each Week:

- **Monday**: Lecture (Background for Labs) - *1016 Rowe*
- **Tuesday**: Lab Session 1 (Starts with Assignment Explanation) - *301A Dunn*
- **Thursday**: Lab Session 2 (Troubleshooting) - *301A Dunn*

Weekly Lab Assignment due before 23:59 following Monday

20% late penalty per day

Attending labs **required** to use TA office hours

No SDAs but your lowest 2 scoring lab assignments will be dropped.

Mid-Term and Final (with review sessions)

Course split into python basics and more applied skills

Week 1: Variables & Data Types
Week 2: Strings, Lists, Sets, & Dictionaries
Week 3: Loops & Conditionals
Week 4: Functions
Week 5: Classes/Object Oriented Programming
Week 6: Files & I/O

Week 7: Study Break
Week 8: Mid-Term

Week 9: Dataframes and Visualisation
Week 10: Probability
Week 11: Regression
Week 12: Machine Learning
Week 13: Future Topics & Final Review
Exam Period: Final

So, let’s start to learn some python

4 basic elements of programming

- **Data**: the **objects** you are actually manipulating
- **Expressions**: computing data values using **data** and **operations**
- Statements: doing something other than compute a value
- **Variables**: storing **data values**

Last term CSCI2202 had 18 students, currently there are 60 students enrolled.

How big an increase, in percent, is this?

- Increase is: `60 - 18`
- Relative increase is: `(60 - 18) / 18`
- In percent: `perc_change = ((60 - 18) / 18) * 100`

Expressions and values

- `((60 - 18) / 18) * 100` is an **expression**
- This expression **evaluates** to `233.33333333333334`
- `60, 18, 100` and `233.33333333333334` are all **values** (i.e., data)
- `/ - * +` are (a subset of) **operations**
- `perc_change` is a **variable** whose value we assign with `=`
- In interactive mode, the python interpreter will automatically print the result of evaluating an expression:

```
>>> ((60 - 18) / 18) * 100
    233.33333333333334
```

```
>>> perc_change = ((60-18)/18)*100
>>> perc_change
    233.33333333333334
```

Assignment is an example of a statement

- A **variable** is a name that is associated with a **value** in the program.
- Expression evaluated then assigned to variable name *(sometimes only actually done when value is needed - "**lazy**")*
- These name–value associations are stored in a "namespace".
- Common to update a variable using its current value so many assignment shortcuts (operator"=")

- Variable name may contain letters, numbers and underscores (but must start with a letter or "_").
- **Reserved words** cannot be used as names.
- Names are **case sensitive**: upper and lower case letters are not the same X != x

```
>>> x = 10_000_000
>>> x
10000000
>>> x = x + 1
>>> x
10000001
>>> x += 1
>>> x
10000002
>>> x -= 1
>>> x
10000001
>>> use_informative_var_names = True
```

Every value has a data **type**

- Value (data) types in python:
  - Integers (`type int`): `0, 1, -3, …`
  - Floating-point numbers (`type float`): `1.0, 0.2, …`
  - Text (a.k.a. “string”, `type str`): `“cool”, ’zero’, “1.03”, …`
  - “Boolean” truth values (`type bool`): `False` and `True`.
  - ...and many more we’ll see later.
- Types determine what we can do with values (and sometimes what the result is).

Every value has a data **type**

The `type` function tells us the type of a value:

```
>>> type(2)
int
>>> type(2 / 3)
float
>>> type("1")
str
>>> type(1 < 0)
bool
```

Types determine how **operators** compute values in **expressions**

Numeric types: integers

- Integers (`type int`) represent positive and negative whole numbers

  (`0, 1, 2, -1, -17, 4096, ...`)

- Values of type `int` have no inherent size limit.
- Can't use "," to "format" integers but can use "_"
  - `1282736` cannot be written as `1,282,736`
  - `1282736` can be written as `1_282_736`.

```
>>> 2 ** (2 ** 2)
16
>>> 2 ** (2 ** (2 ** 2))
65536
>>> 2 ** (2 ** (2 ** (2 ** 2)))
...
```

Numeric types: floats

- Floating-point numbers (`type float`) represent decimal numbers.
- Limited range and precision.
  - Min/max value: $\pm 1.79 \cdot 10^{308}$.
  - Smallest non-zero value: $2.22 \cdot 10^{-308}$.
  - Smallest value > 1: $1 + 2.22 \cdot 10^{-16}$.

(Specific python implementations can have slightly different limits.)

- Floats are approximations (`⅓ != 0.333...4`)
- Can be expressed in scientific notation (`1e-10`)
- Floats also have special values `± inf` (infinity) and `nan` (not a number): `math.inf, math.nan`

```
>>> 2.0 / 4.0
0.5
>>> 1.0 / 3.0
0.3333333333333334
>>> 28.9 ** 2.8
12317.255769964167
>>> 2e18 * 8e-2
1.6e+17
```

Numerical operators

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

https://www.w3schools.com/python/python_operators.asp

These follow standard **PEDMAS** order of operations but if ambiguous use parentheses as you would on a calculator!

Text types: strings

- Strings (`type str`) represent text.
- A string literal is enclosed in single or double quote marks:

```
>>> "Hello world"
'Hello world'
>>> '4" long'
'4" long'
```

- String can contain other types of quote mark, but not the one it starts with.
- More about strings next week!

Types can be converted to other types

- Explicit conversions “**casting**” can often be done using the type name:

```
>>> int(2.0)
```

```
>>> float("−1.05")
```

```
>>> str(0.75 * 1.75)
```

```
>>> type(4 / 2)
<class 'float'>
>>> type(3 * 2.5)
<class 'float'>
>>> type("abc")
<class 'str'>
```

- Conversion from str to number only works if the string contains (only) a numeric literal
- Conversion from int to float is automatic
  - int times/minus/add float becomes a float
  - int divided by int or float becomes a float

Types determine how operators work => expression evaluation

- Same operator can do different things with different types - “**operator overloading**”
- Many languages require you to explicitly say the type of a variable (or can be) - “**static typing**”
- Python works determines the type as it executes each line - “**dynamic typing**”
- Python will try and guess how to convert types as needed.
- Powerful/flexible but can lead to tricky bugs and is harder for the computer to run efficiently.

```
>>> 3 + 4
7
>>> “A” + “B” + “C”
‘ABC’
```

```
>>> x = 5
>>> type(x)
<class 'int'>
>>> type(x / 5)
<class 'float'>
```

Functions allow concise reference to a series of operations

A **function** (aka “procedure/subroutine”) is a named chunk of code.

- The function is called by its **name**
- Defined once but can be called any number of times
- **Operators** are special functions
- Inputs to functions go between ()
- We’ve already used a function type()
- print() is a useful function
- import lets you access large sets of pre-existing functions (see the lab)

```
>>> x = 7
>>> type(x)
<class 'int'>

>>> print(123123 + 2)
123125

>>> print(x)
7

>>> import math
>>> math.log(x)
2.1972245773362196
```

Summary

- Overview of the course: **50% labs**, **50% exams**
- Need to understand any code you submit regardless of where it came from!
- Python code consists of expressions, values, variables, and statements
- integers, floats, bools, and strings are the basic types and what an operator does depends on the types
- Types can be converted and guessed by python - flexible but can be risky
- Functions let you run a piece of code many times
- Import lets you access large libraries of functions written by other people