Midterm Prep

- [ ] Review professors lectures
- [ ] review syllabus for info on each unit
- [ ] Practice writing scripts by hand
- [ ] Practice regex | grep
- [ ] do full emacs tutorial
- [ ] github lecture comparions
- [ ] print
    - [ ] cheat sheets
        - [ ] regexp
        - [ ] emacs shortcuts
        - [ ] shell commands
            - [ ] grep
            - [ ] export
    - [ ] assignments
        - [ ] read and condense all linked docs
    - [ ] relevant method lists
    - [ ] homework solutions
- [ ] look up
    - [x] Abstraction
    - [ ] interpretation steps (REPL)
    - [ ] TR vs grep regexp
    - [ ] why regexp '' and "" is handled differently in regex
    - [ ] locale (collation)
    - [ ] review all discussion slides
    - [ ] 'sys' class
    - [ ] review posix
    - [ ] maybe print eMac's keybindings for diff modes
    - [ ] piazza worksheet
    - [ ] systematically
    - [ ] argeparse

- [ ] organize topics in alphabetical order
- [ ] software components
- [ ] cls instantiation vs self
- [ ] constants/ configuration values as opposed to hard coding
- [ ] child class / parent class / base class
- [ ] hardcoded external filesystem structure
- [ ] sequence indexing — string operations — lists, dictionaries— tuples
- [ ] void element
- [ ] excursion

POSIX means *Portable Operating System Interface for uniX*

eLisp

Interpretation Steps (REPL)

- **read > evaluate > process (putting it into an object / accessing) > loop**
- intrprtr reads objects

Processes everything as a list, operates as framework for expressions

cons () - construct a list, gets handled as a list

Everything in eLisp is being treated as an object

expression is an object

variable > datatype (assign variable) > becomes data object (instance of datatype class)

Evaluating Example :(+ 1 1) , this a list that must be evaluated. broadly speaking: assumes their needs to be a function that need to be

buffer object of datatype integer

Python

Syntax:

Control Flow interpretation: indentation for sequencing steps works in place of (; ,)

Evaluating Expressions:
shell puts it in a bufferspace
regular interpreter: temporary variable

Broad interpreting: no reference to address space,

Evaluating an expression:

Datatype: corresponds to some type of object
evaluating a variable always the value of some object type

Type Casting: No explicit typecasting in python can change

implicit typecasting: int (A and B) * Float D, object conversion

explicit typecasting: int x int y string b float d, convert float to int by typing int (float).

Python Data Types

- **String:**
    - Concatenate (glued together) with ‘+’
    - Repeated with: * (3 * ‘x’ = xxx)
    - Two string literals next to each other are concatenated
    - Substring
        - Slicing [1:4]
- **Numbers:** represented by type str, assign with (=)
    - int
    - float
    - decimal
    - fraction
- **Dictionary** {} keys must immutable (reference pointed to a value)
    - keys changed, hard to access values

    - Cannot use list or dictionary or list
        - need a single variable that doesnt works
    - When you modify a dictionary, you're changing its content in-place, without changing its memory addres
    - Cannot reference a dictionary within a dictionary because its not hashable
        - lose access to innermost dictionary
    - Tuple works because its immutable
- **Tuple**
    - like a list
    - cannot change them, immutable
    - when you dont want data to be changed, assign it a tuple
    - not a KVP, a list. Can have a list inside a tuple

Compound datatypes

- Typecasting in programming languages can be implicit or explicit. Python handles type conversion implicitly while other languages require explicit typecasting.

- Lisp uses lists as a framework to evaluate expressions and treat them as single units. Expressions are treated as functions applied to atomic objects enclosed in lists.

- Python uses indentation instead of curly braces/semicolons to indicate control flow. Expressions are evaluated and may be associated with a temporary memory address.

Memory addresses can store values and data types in linked lists or hash maps depending on the language implementation:

- **Memory addresses don't store the actual value of a variable, but rather reference a location in memory that contains the value and data type.**

- **Lists and other data structures can be stored either as a single address with references to sub-addresses/elements, or using a hash map approach with smaller references under a blanket address.**

- **Dictionaries and other non-hashable types cannot be used as keys**

that map to memory addresses due to self-referencing issues.

- **Programming languages have different approaches to how data is stored and linked via memory addresses, such as linked lists, hash maps, or a single address with nested key-value pairs.**

Hashability

In Python, hashability is a property that determines whether an object can be used as a key in a dictionary or as an element in a set.
Hashable objects have a hash value that remains constant during their lifetime, meaning they need to be immutable. Non-hashable types are those which are mutable and therefore cannot guarantee that their hash value remains the same:

1. **Lists**: Mutable sequences of objects. Lists can be modified after their creation by adding, removing, or changing elements, making them non-hashable.

1. **Dictionaries**: Mutable mappings of keys to values. Like lists, dictionaries can be modified after creation by adding, changing, or removing key-value pairs.

1. **Sets**: Mutable collections of unique elements. Sets in Python can have elements added or removed, thus they are non-hashable.

These types are non-hashable because their content can change over time, which means their hash value can also change. If they were allowed to be used as dictionary keys or set elements, it would be impossible to reliably retrieve or locate them since their hash value at the time of insertion could be different from their hash value when they are accessed later.

The mutability of dictionaries means that if you were able to use a dictionary as a key, changing the dictionary used as a key would potentially change its hash value. This would break the fundamental mechanism by which dictionaries operate, as the dictionary relies on the immutability of keys to maintain a stable, searchable structure.

example: Take dictionary of D and assign it as a value

- python does not allow users to use dicts/lists to be keys
- not hashable, reference is lost
- D wont be able to find for the value is being stored

- not hashable (cant keep track of keys)
    - Dict1={}
    - Dict1[[1,2,3]]=3
    - a=[1,2,3]
    - Dict1[a]=3
    - a=[1,2,4]
    - [1,2,3]->3
    - [1,2,4]->3 doesnt know where 3 is, value of 3 is lost
    - link to reference to value address
    - application of a hash function: assigns and address that references to value

dict keys must be hashable, and dicts themselves are mutable, which means they cannot be used as keys in other dictionaries. The statement d[d] = d attempts to use the dictionary d both as the key and the value, violating the requirement that dictionary keys must be hashable

For contrast, types such as strings, integers, floats, and tuples containing only immutable elements are hashable and can be used as dictionary keys. Python also provides the frozenset type, an immutable version of a set, which is hashable and can be used as a dictionary key.

Memory Allocation

In Python, when you create an instance of a dictionary (or any other object), the Python interpreter allocates a block of memory to store this instance. Each block of memory has a unique address that Python uses internally to reference the object. This memory address is important for various reasons, such as identity comparison and as a basis for the object's hash value when the object is hashable. Here's a closer look at how the memory address works for an instance of a dictionary:

Allocation
When you instantiate a dictionary, Python allocates memory for it. The size of the allocated memory depends on the contents of the dictionary and Python's memory management policies, which include overallocation to minimize the number of reallocations needed as the dictionary grows.

Memory Address
The memory address can be thought of as the dictionary's location in memory. It's a unique identifier that Python uses to access the dictionary. You can use the id() function to see the memory address of any object in Python, including dictionaries:

```
python
my_dict = {}
print(id(my_dict))  # Outputs the unique memory address of
my_dict
```
This address is represented as an integer, and under the hood, it corresponds to the pointer location in memory where the dictionary's data is stored.

Identity and Equality
The memory address of a dictionary instance is crucial for its identity. Two dictionaries may contain the same key-value pairs but be stored at different memory addresses, making them distinct objects:
python
```
dict1 = {'a': 1}
dict2 = {'a': 1}
print(id(dict1) != id(dict2))  # True, because they are stored
at different addresses
print(dict1 == dict2)          # True, because their contents
are equal
```
In this example, dict1 and dict2 are considered equal in terms of their content, but they are distinct objects with different memory addresses, which is why their identities are different.

Implications for Mutability
The memory address of a dictionary plays a role in its mutability. When you modify a dictionary, you're changing its content in-place, without changing its memory address:
python
```
my_dict = {'a': 1}
print(id(my_dict))
my_dict['b'] = 2
print(id(my_dict))  # The memory address remains the same
```
This in-place modification is possible because dictionaries are mutable. The constant memory address despite changes to its content highlights one of the key aspects of mutable objects in Python.

Memory Management
Python's memory management, including garbage collection, uses the memory address to keep track of objects. When an object's reference count drops to zero (meaning no part of your program is using it), Python's garbage collector automatically frees the memory, making it available for other objects.
Understanding how memory addresses work for dictionary instances and other objects in Python is fundamental to grasping concepts like object identity, mutability, and memory management.

TypeCasting

C: A Low-Level, Statically Typed Language

C is designed as a low-level, statically typed language that is closer to the hardware. It provides direct memory access and requires precise control over how data is represented and manipulated in memory. This design choice makes C highly efficient and flexible for system-level programming, but it also imposes certain constraints:

- **Static Typing**: In C, variables are statically typed, meaning their type is determined at compile time and cannot change. This strict type enforcement ensures that the size and layout of data in memory are known, allowing for efficient memory management and execution but requiring explicit typecasting when you want to convert data from one type to another.

- **Manual Memory Management**: Given its low-level nature, C gives programmers direct control over memory allocation and deallocation. Explicit typecasting is often necessary to ensure that operations on pointers and memory access are performed correctly and safely.

- **Performance Optimization**: In contexts where performance and resource utilization are critical, explicit typecasting allows developers to make informed decisions about how operations should be performed, including how and when type conversions happen.

Python: A High-Level, Dynamically Typed Language

Python, on the other hand, is designed as a high-level, dynamically typed language that emphasizes ease of use, readability, and flexibility. It abstracts many of the details related to data types and memory management:

- **Dynamic Typing**: Python variables do not have a fixed type; the type is associated with the value itself, and variables can refer to values of any type. This dynamic typing system allows Python to handle type conversions implicitly, making the language more flexible and reducing the need for explicit typecasting.

- **Automatic Memory Management**: Python manages memory automatically through a garbage collector, which simplifies development by abstracting away the complexities of manual memory management.

This also means that developers don't need to worry about the size and layout of data in memory when performing type conversions.

- **Ease of Use and Productivity**: Python's design prioritizes developer productivity and ease of use. Implicit type conversions align with this goal by enabling developers to write code more quickly and with fewer explicit declarations and type-related boilerplate code.

How Implicit Type Casting Works

Implicit type casting is common in arithmetic operations and comparisons. Python follows a set of rules for type conversion to ensure that operations are performed on compatible data types. Here are some examples of how implicit type casting works:

1. **Arithmetic Operations**: When you perform arithmetic operations with operands of different types, Python automatically converts the operand with lower precision to the type of the higher precision operand.

```
python
# Integer and float operation
result = 3 + 2.5  # Integer is implicitly converted to float
print(result)  # Output: 5.5
```

In this example, the integer **3** is automatically converted to a float (**3.0**) before performing the addition, resulting in a float value (**5.5**).

2. **Boolean in Numeric Contexts**: Python treats True as 1 and False as 0 when used in numeric contexts.

```
python
result = True + 2  # True is implicitly converted to 1
print(result)  # Output: 3
```

Here, **True** is implicitly converted to its numeric equivalent **1**, and then addition is performed.

3. **Comparisons**: During comparisons between different numeric types, implicit type conversion is applied to make them comparable.

```
python
# Comparing an integer and a float
result = 5 < 6.7  # Integer is implicitly converted to float
for comparison
print(result)  # Output: True
```

Limitations

While implicit type casting helps in many situations, it has limitations. Not all types can be automatically converted due to the risk of losing information or because such conversion does not make logical sense. For example, converting from float to integer or from a complex number to any other numeric type cannot be done implicitly because it involves precision loss or is not logically clear. In these cases, Python requires explicit type conversion (casting) by the programmer.

In Python, typecasting a float to an integer can be done using the int() function. This function converts a floating-point number to an integer by removing the decimal part of the number, effectively truncating towards zero. It's important to note that this conversion does not round the number but simply discards the fractional part.
Here's how you can typecast a float to an integer:

```
python
my_float = 3.14
my_int = int(my_float)
print(my_int)  # Output will be 3
In this example, the float 3.14 is converted to the integer 3.
```

- **Truncation Towards Zero**: The conversion always truncates towards zero, which means it removes the fractional part and moves towards zero to determine the integer value.
- **No Rounding**: The int() function does not round the number to the nearest integer; it simply truncates the decimal part. If rounding is desired, you should use the **round()** function before converting to an integer.
- **Explicit Conversion**: This is an example of explicit typecasting, where the programmer specifies the conversion to be made. It's a straightforward and readable way to convert data types in Python.

By using the int() function, you can easily convert floats to integers when you need to perform operations that require integer values or when the fractional part of a number is not needed.

Conclusion

The need to explicitly change data types in C but not in Python reflects the different goals and design philosophies of the two languages. C provides low-level access and control, requiring explicit typecasting for precise data type management and performance optimization. Python, aiming for high-level abstraction and ease of use, handles many type conversions implicitly, allowing developers to focus on the logic of their programs without getting bogged down by the details of data types and memory management. This difference illustrates the trade-off between control and convenience that characterizes the choice between low-level and high-level programming languages.

Implicit type casting, also known as automatic type conversion, occurs in Python when the interpreter automatically converts one data type to another without any explicit instruction from the user. This feature is designed to increase the readability and maintainability of the code by handling common type conversion operations automatically. Implicit type casting happens during operations involving mixed data types, where Python converts the data type with lower precision to the one with higher precision to avoid data loss.

Additional Examples

1. **Converting a Positive Float**:

```
python
positive_float = 7.99
positive_int = int(positive_float)
print(positive_int)  # Output will be 7
```

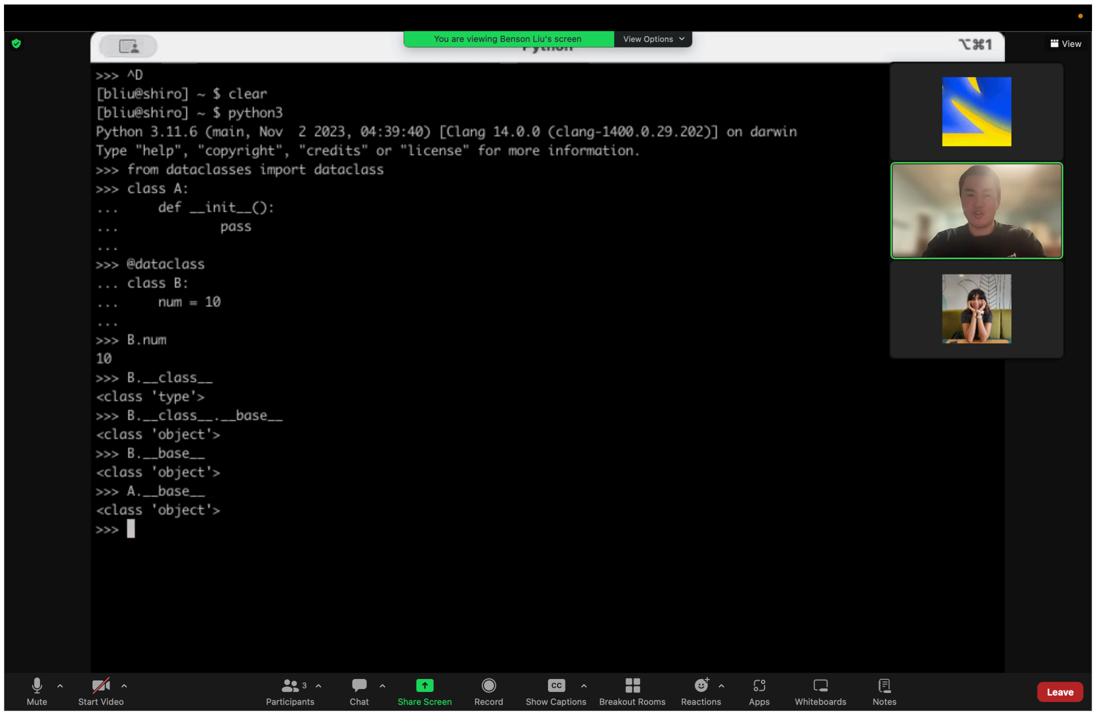
2. **Converting a Negative Float**:

```
python
negative_float = -2.99
negative_int = int(negative_float)
print(negative_int)  # Output will be -2
```

In both cases, the int() function removes the fractional part of the number, truncating towards zero.

Conclusion

Implicit type casting simplifies many operations in Python by automatically handling data type conversions. It allows for cleaner code by avoiding the need for explicit type conversions in many common scenarios. However, programmers must be aware of the rules and limitations of implicit type casting to avoid unexpected behavior or errors in their programs.



```
>>> ^D
[bliu@shiro] ~ $ clear
[bliu@shiro] ~ $ python3
Python 3.11.6 (main, Nov  2 2023, 04:39:40) [Clang 14.0.0 (clang-1400.0.29.202)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from dataclasses import dataclass
>>> class A:
...     def __init__():
...             pass
...
>>> @dataclass
... class B:
...     num = 10
...
>>> B.num
10
>>> B.__class__
<class 'type'>
>>> B.__class__.__base__
<class 'object'>
>>> B.__base__
<class 'object'>
>>> A.__base__
<class 'object'>
>>> 
```

BASH

export

The export PATH command in Unix-like operating systems is used to set or modify the PATH environment variable. This variable tells the shell where to look for the executables (commands or programs) that you try to run. When you type a command in the terminal, the shell searches the directories listed in your PATH environment variable to find the executable for that command.
Here's a breakdown of what happens when you use the export PATH command:

- **export**: This is a shell builtin command used to set or modify environment variables for the current shell session and any subprocesses or child sessions launched from it. Environment variables set without export are not inherited by child processes.

- **PATH**: This is an environment variable that specifies a list of directories separated by colons (:) in Unix-like systems. When you run a command, the shell looks through these directories in the order they are listed in PATH to find the executable file for the command.

- **Modifying PATH**: You can modify the PATH variable to include additional directories where the shell should look for executables. For example, if you have scripts or programs in a directory not included in the PATH, you can add that directory to the PATH variable. This is done by appending or prepending that directory to the existing PATH value.

Here's an example command that appends a new directory (/usr/local/my_program/bin) to the existing PATH:

```
sh
export PATH=$PATH:/usr/local/my_program/bin
This command ensures that in addition to the existing
directories, the shell will now also search in /usr/local/
my_program/bin for executable files.
```

And here's how you might prepend a directory to the PATH, making it the first directory that the shell searches:

```
sh
export PATH=/usr/local/my_program/bin:$PATH
```

Using export PATH like this affects the current shell session and any programs started from it after the export command is run. It does not permanently change the PATH for future sessions. To make changes permanent, you would typically add the export PATH command to a shell initialization file like ~/.bashrc, ~/.bash_profile, or ~/.profile, depending on the shell and operating system.

Control Flow

Python

Indentation in Python is not just a matter of style; it's a significant part of the syntax that determines the structure and flow of the code. Here's why indentation matters in Python, particularly in the context of control flow:

1. **Defines Code Blocks**: Unlike many other programming languages that use braces {} to define blocks of code, Python uses indentation. The level of indentation indicates which statements belong to which block, making it visually clear how the code is organized.

2. **Determines Execution Flow**: The indentation level of a statement determines its relationship to the control flow constructs (if, elif, else, for, while, try, except, etc.). For example, all the statements within the same block of a conditional or loop must be indented at the same level to be

executed as part of that construct.

3. **Enforces Readability**: By enforcing indentation as a part of the syntax, Python encourages readable and consistently formatted code. This readability is especially crucial for control flow, where the logic can become complex. Proper indentation helps developers quickly understand the structure and flow of the code, including which blocks of code are executed under certain conditions or as part of loops.

4. **Syntax Errors**: Incorrect indentation can lead to IndentationError or SyntaxError, causing the program to fail to execute. For instance, if the lines of code within a loop are not correctly indented, Python will not recognize them as part of the loop block, leading to errors or unexpected behavior.

5. **Nesting Control Structures**: In complex control flows, structures may be nested within each other (e.g., an if statement inside a for loop). Indentation visually delineates the levels of nesting, making it clear which control structure each code block belongs to.

Type Checking

1. **Static Type Checking (compiling)**:
    - In statically typed languages, data types are checked at compile-time, before the program is executed.
    - The compiler enforces strict rules about variable and function parameter types. If a variable is declared as an integer, it must always hold an integer value, and the compiler ensures this.
    - Static type checking helps catch type-related errors before the program is run, reducing the likelihood of runtime errors.
    - Examples of statically typed languages include C, C++, Java, and Swift.
2. **Dynamic Type Checking (interpreting)**:
    - In dynamically typed languages, data types are checked at runtime, while the program is executing.
    - Variables can change their data type during the execution of the program without explicit declarations.
    - This flexibility allows for more dynamic and expressive code but can lead to runtime errors if data types are used incorrectly.
    - Examples of dynamically typed languages include Python, JavaScript, Ruby, and PHP.

Here's a comparison between the two approaches:

- **Error Detection**:
    - Static Type Checking: Errors are detected at compile-time, which means that you can catch them before running the program.
    - Dynamic Type Checking: Errors may only be detected at runtime, potentially causing unexpected behavior during execution.
- **Code Flexibility**:
    - Static Type Checking: More restrictive, as you need to specify types explicitly. This can make the code less flexible but more predictable.
    - Dynamic Type Checking: Allows for more flexibility, as variables can change their types on the fly. This can lead to concise and expressive code but can also introduce potential issues if not managed carefully.
- **Ease of Use**:
    - Static Type Checking: Requires more type annotations and declarations, which can make the code longer and more verbose.
    - Dynamic Type Checking: Generally requires fewer type annotations, leading to more concise code.
- **Performance**:
    - Static Type Checking: Can often lead to more optimized and faster code because the compiler has more information about types at compile-time.
    - Dynamic Type Checking: May incur some runtime overhead due to type checks during execution.

    - LA question split up
      a b c d e
    - week 1 week 2 week 3 week 4 week 5

grep, tr, com

combination with grep and regex

difference between regex and extended regex (without -E)

- have to escape the special characters