Announcements

- Homework 1 has been posted!
  - Click the link to add your private repository, and then download to your system.
  - Due at midnight on Friday
  - Make sure you change the appropriate spot in the README to DONE when you are finished so I know it is ready to grade.
    - Otherwise it will count against your available late days!
- Polling during lecture starts today!
  - `rembold-class.ddns.net`
- CS Tea
  - Thursdays in Ford 2nd floor, 11:30 - 12:30
  - Tea, cookies and (usually) pizza!

Review Question

When would the python interpreter generally catch the error in the expression below?

```
"They" + " are " + 21 + " years old."
```

A) Before the code is run
B) While the code is run
C) It will not catch the error, but one exists
D) There is no error in this code

Programming Frameworks

- Low vs High level:
  - Low: language uses primitives that operate on the machine level (write 10 bits of data to this location)
  - High: language uses more abtract primitives (open a window and print a message to it)

Programming Frameworks

- Low vs High level:
  - Low: language uses primitives that operate on the machine level (write 10 bits of data to this location)
  - High: language uses more abtract primitives (open a window and print a message to it)
- General vs Targeted:
  - General: Can be used across many different disciplines and for varied tasks
  - Targeted: Excels at one or a few different use cases

Programming Frameworks

- Low vs High level:
  - Low: language uses primitives that operate on the machine level (write 10 bits of data to this location)
  - High: language uses more abtract primitives (open a window and print a message to it)
- General vs Targeted:
  - General: Can be used across many different disciplines and for varied tasks
  - Targeted: Excels at one or a few different use cases
- Interpreted vs Compiled:
  - Interpreted: source code executed directly by interpreter
  - Compiled: source code first converted to machine code by compiler and then instructions executed from that code

Programming Frameworks

- Low vs High level:
  - Low: language uses primitives that operate on the machine level (write 10 bits of data to this location)
  - High: language uses more abtract primitives (open a window and print a message to it)
- General vs Targeted:
  - General: Can be used across many different disciplines and for varied tasks
  - Targeted: Excels at one or a few different use cases
- Interpreted vs Compiled:
  - Interpreted: source code executed directly by interpreter
  - Compiled: source code first converted to machine code by compiler and then instructions executed from that code
- Different language fall on different portions of these spectrum

Where Python stands

- A fairly high level language (you don't have to worry about object sizes or where in memory they are being stored)

Where Python stands

- A fairly high level language (you don't have to worry about object sizes or where in memory they are being stored)
- Pretty general. Python is used in a massive range of applications these days.

Where Python stands

- A fairly high level language (you don't have to worry about object sizes or where in memory they are being stored)
- Pretty general. Python is used in a massive range of applications these days.
- Definitely interpreted

Where Python stands

- A fairly high level language (you don't have to worry about object sizes or where in memory they are being stored)
- Pretty general. Python is used in a massive range of applications these days.
- Definitely interpreted
  - Cost is that things can run a bit slower

Where Python stands

- A fairly high level language (you don't have to worry about object sizes or where in memory they are being stored)
- Pretty general. Python is used in a massive range of applications these days.
- Definitely interpreted
  - Cost is that things can run a bit slower
  - Pro is that debugging is generally easier and more intuitive

Where Python stands

- A fairly high level language (you don't have to worry about object sizes or where in memory they are being stored)
- Pretty general. Python is used in a massive range of applications these days.
- Definitely interpreted
  - Cost is that things can run a bit slower
  - Pro is that debugging is generally easier and more intuitive
- Other weaknesses:

Where Python stands

- A fairly high level language (you don't have to worry about object sizes or where in memory they are being stored)
- Pretty general. Python is used in a massive range of applications these days.
- Definitely interpreted
  - Cost is that things can run a bit slower
  - Pro is that debugging is generally easier and more intuitive
- Other weaknesses:
  - Generally has weak static semantic checking, which can be problematic for highly reliability constraints or complex, long projects

Where Python stands

- A fairly high level language (you don't have to worry about object sizes or where in memory they are being stored)
- Pretty general. Python is used in a massive range of applications these days.
- Definitely interpreted
  - Cost is that things can run a bit slower
  - Pro is that debugging is generally easier and more intuitive
- Other weaknesses:
  - Generally has weak static semantic checking, which can be problematic for highly reliability constraints or complex, long projects
- Other Pros:

Where Python stands

- A fairly high level language (you don't have to worry about object sizes or where in memory they are being stored)
- Pretty general. Python is used in a massive range of applications these days.
- Definitely interpreted
  - Cost is that things can run a bit slower
  - Pro is that debugging is generally easier and more intuitive
- Other weaknesses:
  - Generally has weak static semantic checking, which can be problematic for highly reliability constraints or complex, long projects
- Other Pros:
  - Simple syntax makes fairly easy to learn and read

Where Python stands

- A fairly high level language (you don't have to worry about object sizes or where in memory they are being stored)
- Pretty general. Python is used in a massive range of applications these days.
- Definitely interpreted
  - Cost is that things can run a bit slower
  - Pro is that debugging is generally easier and more intuitive
- Other weaknesses:
  - Generally has weak static semantic checking, which can be problematic for highly reliability constraints or complex, long projects
- Other Pros:
  - Simple syntax makes fairly easy to learn and read
  - Huge selection of freely available libraries makes it incredibly flexible

Turtle Power

- Python programs are frequently called scripts
  - Sequence of definitions and commands the computer should run
  - Filename ends in `.py` suffix
- Scripts are executed by the Python interpreter in a shell
  - When you run a script a new shell will be started and used automatically
  - You can also launch just a shell to be able to type Python commands in directly
- Can access a shell through Anaconda by launching Anaconda Prompt and then typing `python`
  - You know you are in the shell because a shell prompt will be at the start of each line: >>>

Not really my type

- Programs manipulate data objects
- Objects come in two varieties:
  - Scalar, which can not be broken down into smaller chunks
  - Non-scaler, which have some internal structure (are made up of smaller chunks)
- Objects also have a type that defines how a program can interact with them
  - This object is a human, so it can walk, talk, learn, etc
  - This object is an integer so it can be added, subtracted, multiplied, etc

Types of Scalar Objects

- 4 types
  - `int` - represents integers. (`5,10,123`)
  - `float` - represents real numbers (`3.14, 2.718`)
  - `bool` - represents boolean values (`True, False`)
  - `NoneType` - has only one value: `None`
- Can check an objects type using, conveniently, `type()`

Types of Scalar Objects

- 4 types
  - `int` - represents integers. (`5,10,123`)
  - `float` - represents real numbers (`3.14, 2.718`)
  - `bool` - represents boolean values (`True, False`)
  - `NoneType` - has only one value: `None`
- Can check an objects type using, conveniently, `type()`

```
>>> type(3.14159)
float
>>> type(789)
int
```

Changing and Comparing

- You can change an object to a certain type by using that type's keyword
  - `float(3)` will result in `3.0`, a float
  - `int(3.85)` will *truncate* to `3`, an integer
  - `int(True)` will result in `1`, an integer
  - You can't convert `NoneType` objects
- Python will frequently try to guess and do these conversions for you if needed
- You can compare expressions to see if they evaluate to the same value
  - `==` will check if the expressions give the same value
  - `!=` will check if the expressions give different values

Integer and Float Operations

- `i + j` - the sum of `i` and `j` *
- `i - j` - the difference between `i` and `j` *
- `i * j` - the product of `i` and `j` *
- `i // j` - the integer division of `i` by `j` *
- `i / j` - the division of `i` by `j` †
- `i % j` - the remainder when `i` is divided by `j` *
- `i ** j` - `i` to the power of `j` *

---

* - Returns `int` if both `i` and `j` integers, `float` otherwise

† - Returns `float` always

Orders of Operations!

- Basic order of operations applies just like in math!
- Operations in parentheses done first
- Without parentheses, order of operations proceeds as
  - `**`
  - `*`
  - `/`
  - `+` and `-` executed left to right

Boolean Operations

You have 3 basic operators you can apply to booleans:

- `a and b` - True if both `a` and `b` are True, False otherwise
- `a or b` - True if at least 1 of `a` and `b` is True, False otherwise
- `not a` - True if `a` is False, False if `a` is True

```
>>> True and False
False
>>> True or (False and True)
True
>>> True and not False
True
```

Understanding Check

What is the resulting type of the below expression?

```
3.15 * int(False) + (int(10.0) // 2)
```

A) Integer
B) Float
C) Boolean
D) NoneType

Binding Variables

- Frequently have a value, as output by an expression, that we want to use later
- Use = to assign a value to a variable name

```
pi = 3.14159
```

- Stores the value in memory, which can be retrieved by typing the given name `pi`.

Binding Variables

- Frequently have a value, as output by an expression, that we want to use later
- Use = to assign a value to a variable name

```
pi = 3.14159
```

- Stores the value in memory, which can be retrieved by typing the given name `pi`.

Warning!

You use = to assign a value to a variable name. You use == to compare two values. These are easy to mix up, and can lead to some weird errors in some cases!

Why Bother?

- Good variable names can remind us what a value represents
- Descriptive names can make checking code or debugging errors easier to spot

```
pi = 3.14159
radius = 5
area = pi * radius ** 2
```

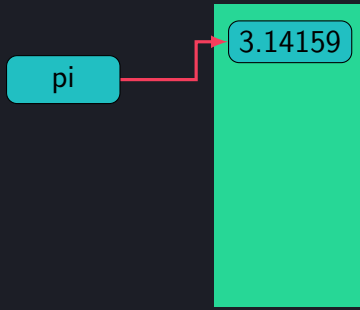
- If a value is reused many times throughout a piece of code, binding it to a variable means you only have to change it in *one* place if you must later

```
fav = 8
print(fav)
print(fav ** 2)
```

All things must change

- You can rebind variables using a new assignment statement
- Old value gets lost
- Variables only change upon assignment. Changing something that a variable depends on does *not* update that variable.

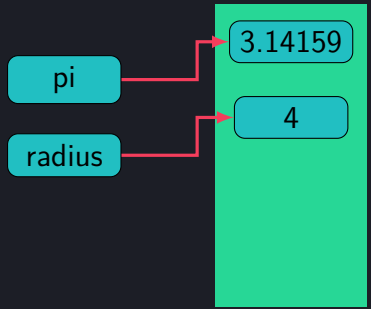
```
pi = 3.14159
radius = 4
circ = 2 * pi * radius
radius = radius + 2
```



All things must change

- You can rebind variables using a new assignment statement
- Old value gets lost
- Variables only change upon assignment. Changing something that a variable depends on does *not* update that variable.

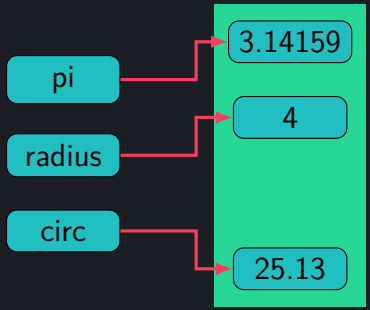
```
pi = 3.14159
radius = 4
circ = 2 * pi * radius
radius = radius + 2
```



All things must change

- You can rebind variables using a new assignment statement
- Old value gets lost
- Variables only change upon assignment. Changing something that a variable depends on does *not* update that variable.

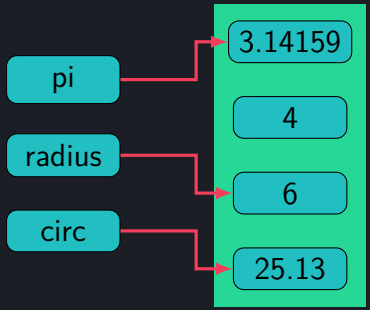
```
pi = 3.14159
radius = 4
circ = 2 * pi * radius
radius = radius + 2
```



All things must change

- You can rebind variables using a new assignment statement
- Old value gets lost
- Variables only change upon assignment. Changing something that a variable depends on does *not* update that variable.

```
pi = 3.14159
radius = 4
circ = 2 * pi * radius
radius = radius + 2
```



Variable Names

There are some important things to keep in mind when coming up with variable names:

- Capitalization matters! `radius` and `Radius` are different
- Variables can have numbers in the name, but they can't *start* with a number
- Underscores are the only acceptable special character
- There is a small set of special python keywords that you can't use for variables.
  - My general recommendation is, if the name gets highlighted when you type it because the IDE thinks it is important, choose a different name
- Make your variable names meaningful!