Python standard library
Introduction to Modules
Try/Except

Lesson Learning objects

By the end of the lesson, you will be able to:

1. Import Python modules
2. Use the try/except structure for error handling
3. Identify common Python error types
4. Implement simple exception handling
5. Write clear, well-commented, and properly documented code

Review

Write a Python program that asks the user for a gene name and then asks the user for the number of nucleotides in its coding sequence. And print as follows:

Your sequence name is: Test
Please enter the length of the sequence: 9
The length of the DNA sequence is: 9.0

Importing Python Modules

Importing Modules in Python

- To use a module, you must first **import** it.

```
import math    # see below
```

  - Put this at the very top of the program
- Then your program can use the things in the module
  - Their names are `module.name`
  - So `math.log` (function) and `math.pi` (constant)
  - You call functions with parentheses
    - Just like `input()` and `print()`
  - Each function has its own rules about what its arguments are, what they mean, how many there are, etc
- Any issues with importing everything? `import math`
- Only import the libraries you need!
  - Do this for documentation, for efficiency, for style, to avoid collisions
  - But how?

A Variation on Import

- You can instead import particular functions or constants specifically by writing import this way:

```
from math import sqrt, sin, cos, tan, pi
```

  – List the names that you are importing, separated by commas
  – Then you can use them without the "`math.sin`" or "`math.`**`radius`**"

```
y = sin(angle) * radius
```

  – Saves typing if you use a function many times

Import everything: Avoid This Style

- You will come across this style of `import`:

```
from math import *
```

  - But once again, this imports everything in the module
  - And you don't have to use "`math.pi`"

```
num = e ** pi
```

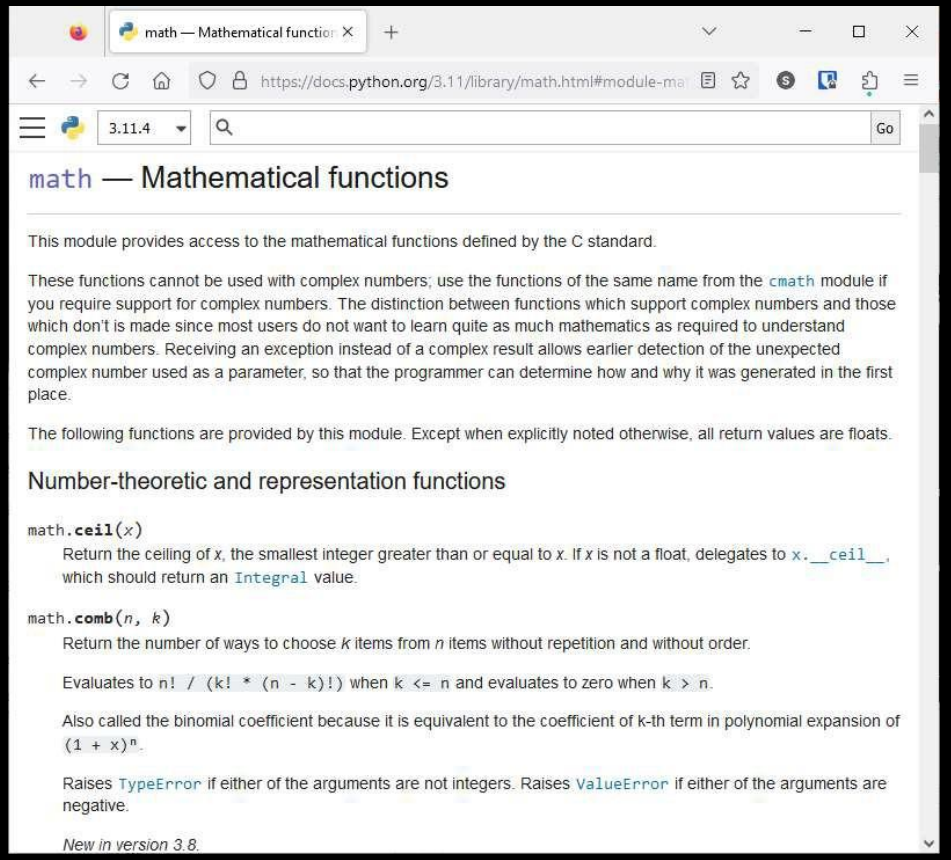
  - Sounds great, right? There can be a catch…
  - What if next version of Python adds a new function which is the same name as one of your variables or functions?
  - Your code could break!
  - Good programmers avoid "`from module import *`"

  - Go over the next slide on your own for more information on why to avoid this style

Finding functions in a package

```
import math
help(math)
```

```
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module provides access to the mathematical functions defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x. The result is between 0
        and pi.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x. The result is between -
        pi/2 and pi/2.
```



The `try` / `except` Structure

Common Python Error Types

- **`NameError`** is thrown when an object could not be found
- **`IndentationError`** is thrown when there is an incorrect indentation
- **`SyntaxError`** is thrown by the parser when a syntax error is encountered
- **`TypeError`** is thrown when an operation or function is applied to an object of an inappropriate type
- **`ValueError`** **is thrown when a function's argument is of an inappropriate type**
- **`IndexError`** is thrown when trying to access an item at an invalid index
- **`ModuleNotFoundError`** is thrown when a module could not be found
- **`KeyError`** is thrown when a key is not found
- **`ImportError`** is thrown when a specified function can not be found
- **`StopIteration`** is thrown when the `next()` function goes beyond the iterator items
- **`ZeroDivisionError`** is thrown when the second operator in the division is zero

- There are many more! https://docs.python.org/3/library/exceptions.html

Start out by learning these!

The `try` / `except` Structure

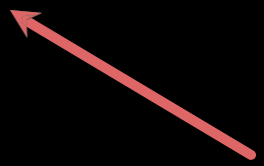
- Before we get into why exception handling is essential and types of built-in exceptions that Python supports, it is necessary to understand that there is a subtle difference between an **error** and an **exception**
- Errors cannot be handled, while Python exceptions can be handled at the run time
- An Error might indicate critical problems that a reasonable application should not try to catch, while an Exception might indicate conditions that an application should try to catch
- You surround a dangerous section of code with `try` and `except`
- If the code in the `try` works - the `except` is skipped
- If the code in the `try` fails - it jumps to the `except` section
- It get's a little more complex than above, but that's a good start, but let's dig deeper

Why Exception Handling?

- Even if the syntax of a statement or expression is correct, it may still cause an an exception to be thrown when the program is executed
- So, we need to write code that catches and deals with exceptions that arise while the program is running
- i.e., “Do these steps, and if any problem crops up, handle it this way.”
- This approach obviates the need to do explicit checking at each step in the algorithm

```
$ cat notry.py
astr = 'Hello Bob'
istr = int(astr)
print('First', istr)
astr = '123'
istr = int(astr)
print('Second', istr)
```

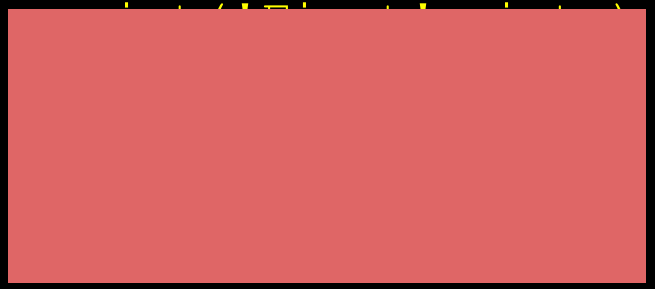
```
$ python3 notry.py
Traceback (most recent call last):
File "notry.py", line 2, in <module>
istr = int(astr)
ValueError: invalid literal for int() with
base 10: 'Hello Bob'
```



All
Done

The program stops here

```
$ cat notry.py
astr = 'Hello Bob'
istr = int(astr)
```



```
$ python3 notry.py
Traceback (most recent call last):
File "notry.py", line 2, in <module>
istr = int(astr)
ValueError: invalid literal for int() with
base 10: 'Hello Bob'
```

All Done

Simple Exception Handling

- The `try` statement has the following form:

```
try:
 <body>
except <ErrorType>:          # ErrorType is optional
 <handler>
```

```
try:
 <body>
except ErrorTypr as err:
 <statements that can refer to err>
```

- When Python encounters a `try` statement, it attempts to execute the statements inside the body
- If no error, control passes to the next statement after the `try…except`
- Let's see the simplest form, ***and then*** see how Python can look for an **`except`** clause with a **`matching error type`**

- `NameError` is thrown when an o
- `IndentationError` is thrown w
- `SyntaxError` is thrown by the p
- `TypeError` is thrown when an o
- `ValueError` is thrown when a
- `IndexError` is thrown when tryi
- `ModuleNotFoundError` is throv
- `KeyError` is thrown when a key
- `ImportError` is thrown when a
- `StopIteration` is thrown when
- `ZeroDivisionError` is thrown

```
astr = 'Hello Bob'
try:
    istr = int(astr)
except:
    istr = -1

print('First', istr)

astr = '123'
try:
    istr = int(astr)
except:
    istr = -1

print('Second', istr)
```

When the first conversion fails - it just drops into the except: clause and the program continues.

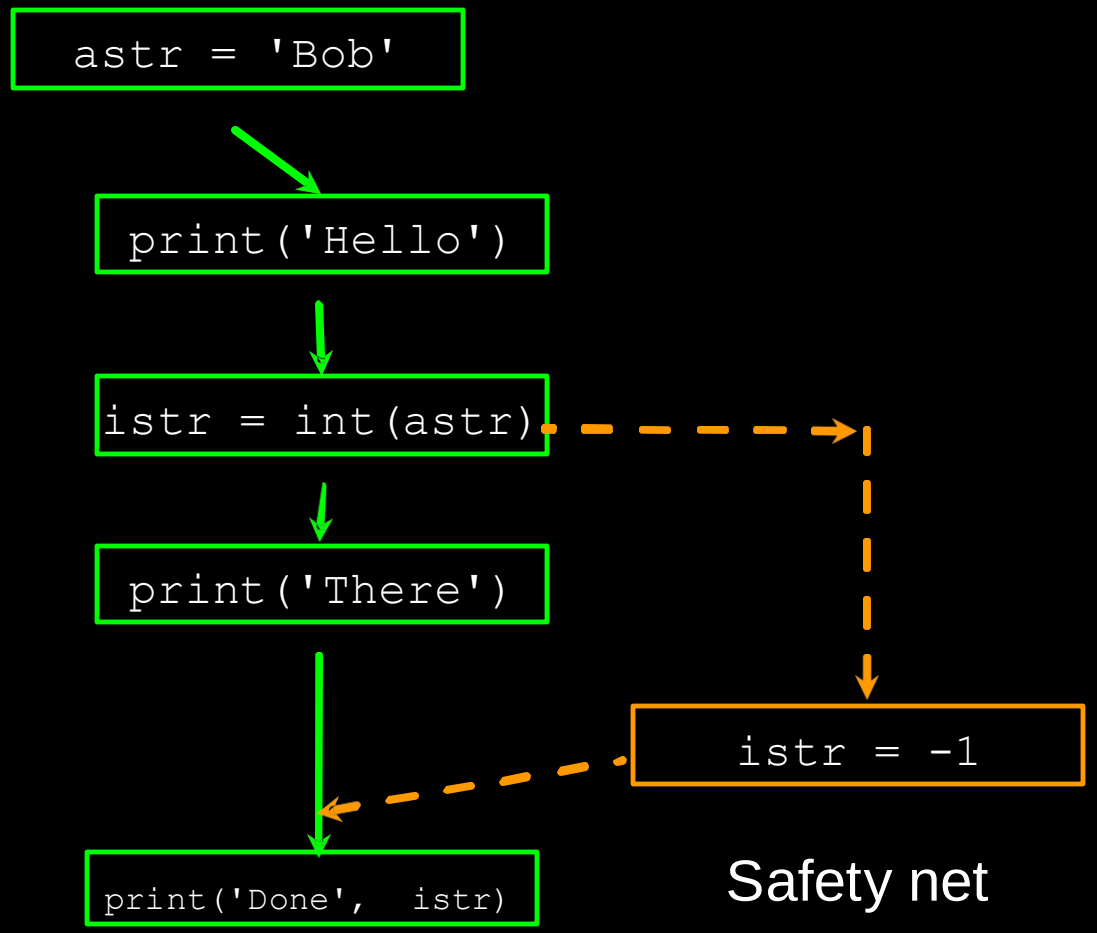
```
$ python tryexcept.py
First -1
Second 123
```

When the second conversion succeeds - it just skips the except: clause and the program continues.

try / except

```
astr = 'Bob'
try:
    print('Hello')
    istr = int(astr)
    print('There')
except:
    istr = -1

print('Done', istr)
```

Hello
Done -1



Updated by CL

Sample `try` / `except`

```
rawstr = input('Enter a number:')
try:
    ival = int(rawstr)
except:
    ival = -1

if ival > 0 :
    print('Nice work')
else:
    print('Not a number')
```

```
$ python3 trynum.py
Enter a number:42
Nice work
$ python3 trynum.py
Enter a number:forty-two
Not a number
```

Your turn

```
try:
    result = 10 / 0
except:
    print("Error: Division by zero is undefined in mathematics and cannot be processed.")
```

What will happen when running this code?

try / except

```
try:
    result = 10 / 0
except ZeroDivisionError as err:
    print("Error:", err)
```

```
Error: division by zero
```

```
try:
    apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
    name = apes[5]
except IndexError as err:
    print("Error:,", err)
```

```
Error:, list index out of range
```

Your turn

```
try:
    my_dna = int("CTGATCGATTAC")
except ……. as err:
    print("You got Error:", err)
```

```
You got Error: invalid literal for int() with base 10: 'CTGATCGATTAC'
```

What will be the type of the error?

Summarizing

- The `try` statement works as follows.
  - First, the ***try clause*** (the statement(s) between the `try` and `except` keywords) is executed
  - If no exception occurs, the ***except clause*** is skipped and execution of the `try` statement is finished
  - If an exception occurs during execution of the ***try clause***, the rest of the clause is skipped. **Then if its type matches the exception named after the** `except` keyword, the ***except clause*** is executed, and then execution continues after the `try` statement
  - If an exception occurs which **does not match the exception** named in the except clause, it is passed on to outer `try` statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above

https://docs.python.org/3/tutorial/errors.html

Comments

Code is for humans

- So far, we have been writing **code for the computer.**
  (We tried to make the computer do something.)
- **Programs are meant to be read by humans and only incidentally for computers to execute.**
  (Harold Abelson, *Structure and Interpretation of Computer Programs*)

- How so?
  - Programmers spend more time reading someone else's code than writing their own.
  - There's no such this as a one-off script.
    (If you have saved a script as a file, it's no longer one-off, and you or someone else will eventually use it again and want to adapt it.)

- Readability, documentation, and maintainability very important.

How do you code for humans?

- Write clear code.
- Comment your code.
- Document your code.

1. Write clear code:general principles

- **KISS: Keep It Simple, Stupid!**
  - Do not write code that is more complicated than necessary.
  - It will take too long for the next programmer of your future self to decode.
  - Your cleverest code will need someone cleverer than you to debug.

- **DRY: Don't repeat yourself.**
  Use functions to extract repeated code.
  - Less code to figure out, less possible bugs, less code to maintain.

- **Separation of concerns**
  Each function should do only one thing, and do it well.
  This makes it easier to figure out, bugs become less complex, documentation becomes easier to write, and code can be reused in more situations (DRY).

1. Write clear code:general principles

A couple of code style guidelines can help too:

- Clear variable and function names

  b □ calls_per_postal_code

- One statement per line
- Use a consistent style
- Prefer small functions over long ones (as long as they perform a non-trivial task).
- Don’t reinvent the wheel; use existing functions and packages.
- No cleverness.
- Use comments and documentation

2. Comment your code

- Comments start with #; anything after that is a comment.

```
# store the DNA sequence in a variable
my_dna = "ATGCGAGT"
# calculate the length of the sequence and store it in a variable
dna_length = len(my_dna)
# print a message telling us the DNA sequence lenth
print("The length of the DNA sequence is " + dna_length)
```

- Keep comments succinct.
- Describe ***why*** your the code is doing something.
- Describe on a high level what parts of the code are doing.

```
# Get to user input an integer
while True:
  try:
    input_string = input("Enter an integer a=")
    a = int(input_string)
    break
  except:
    print("Not an integer; try again")
# Determine if integer a is prime
a_is_prime = all(a%i for i in range(2,a))
  # brute force determination of whether
  # a is prime (was easier to code than
  # a more sophisticated algorithm).
# Report back result to screen
print("a is prime?", a_is_prime)
```

3. Document your code
Python Docstrings

- You 're familiar with “#” to indicate comments explaining what’s going on in a Python file
- Python has a special kind of commenting convention: documentation strings or "docstrings" to document a function
- These go inside the function definition, as the very first thing
- So it must be indented!
  - It starts with """ (three double quotes)
  - Continues across multiple lines until another """

Example Document String

```
"""
Calculate the percentage of G and C character in base_seq
"""
seq = 'ATCCGGGG'
base_seq = (seq.count('G') + seq.count('C')) / len(seq)
```

```
from random import randint

def replace_base_randomly_using_names(base_seq):
    """Return a sequence with the base at a randomly selected position of base_seq replaced
    by a base chosen randomly from the three bases that are not at that position"""
    position = randint(0, len(base_seq) - 1)       # -1 because len is one past end

    base = base_seq[position]
    bases = 'TCAG'
    bases.replace(base, '')                        # replace with empty string!
    newbase = bases[randint(0,2)]
    beginning = base_seq[0:position]               # up to position
    end = base_seq[position+1:]                    # omitting the base at position
    return beginning + newbase + end
```

Example Document String

```
def triangle(size, letter):
    """
    NoneType : triangle(size, letter)
    Takes: int in the first argument (size of the triangle)
           string (character) in the second argument
    Returns: NoneType
    e.g. triangle(2, "+")
    """
    for i in range(1, size+1):
        print(letter * i)

triangle(2, "+")
```

2nd Type Document String

```
def triangle(size, letter):
    """
    @param size: int in the first argument (size of the triangle)
    @param letter: string (character) in the second argument
    @return: NoneType
    """
    for i in range(1, size+1):
        print(letter * i)

triangle(2, "+")
```

Since Moving to PyCharm, I now use this style, and PyCharm helps to write them!

We'll use this form in the course!