INFO5002: Intro to Python for Info Sys

Week 6

Slides created by: Zachary Doucet

Northeastern University

Week 6

I. Encapsulation

II. Modules and Packages

III. Error Handling

IV. Debugging

Admin

Key dates

- Final exam Dec 12th 9-12 am same class room.
- Project Dec 14 at 11:59 pm.
- Pizza party after class today from 1-2:30 pm.

Recap

Introspection to verify type

- type

```
x = "orange"
type(x)
```

```
x = 123
type(x)
```

```
x = 3.14
type(x)
```

- callable

```
x = "banana"
callable(x)
```

```
def add(x, y):
    return x + y
x = add
callable(x)
```

```
if type(x) is str:
    print("I am string")
elif type(x) is int:
    print("I am int")
elif type(x) is float:
    print("I am float")
```

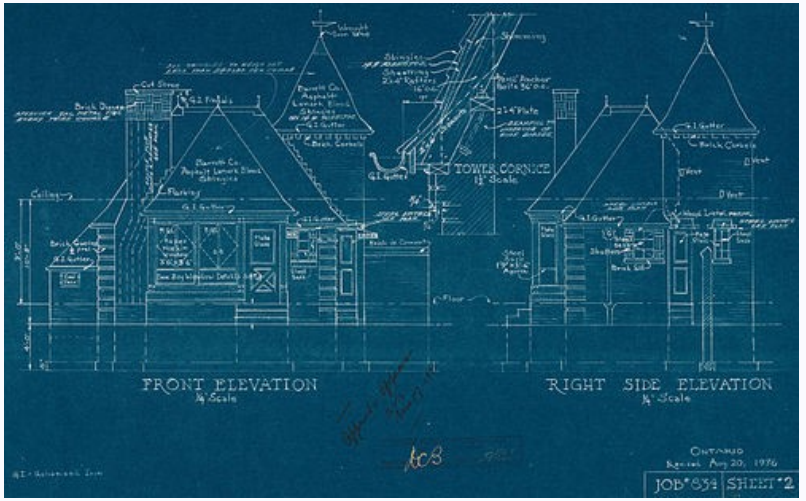
Source: Wikimedia

Classes to group functionality

- Functions group operations together and collections group data together.
- Classes allow you to collect functions and variables together.

```
class Car:
    ...
```

```
car = Car()
```



Source: City of Toronto Archives

The holy grail

“Everything is an object. Treat everything like an obj

Source: Monty Python and the Holy Grail

Inheritance

- Inheritance is a way to save code by copying all the **methods and attributes** from the parent to the child.

```
class Animal:
    def __init__(self, name):
        self.name = name
```

```
animal = Animal("Timmy")
print(animal.name)
```

```
class Dog(Animal):
    def woof():
        print("Woof!")
```

```
dog = Dog("Timmy")
print(dog.name)
Dog.woof()
```

Polymorphism

- Super-class group of objects can all execute a method from the same super-class.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("Squeek")
```

```
class Dog(Animal):
    def speak(self):
        print("Woof!")
```

```
class Cat(Animal):
    def speak(self):
        print("Meow.")
```

```
x = Animal()
y = Dog()
z = Cat()
x.speak()
y.speak()
z.speak()
```

Encapsulation

I want to protect my data!

- It is generally good practice to expose only the **minimum** amount of data necessary.

```
class BankAccount:
    id = 0
    name = ""
    balance = 0
    credit = 0
    interest_rate = 0
    overdrafted = False
    login_dates = []
```

Maybe don't need to have all of these attributes **publicly** exposed.

You can hide with encapsulation

- Encapsulation is the process of selectively hiding data between components.
  - Protect from unauthorized or accidental mutations.
  - Add validation to getting and mutating.
  - Hiding business logic.

There are three access modifiers

- Public: any component can access (default).
- Protected: class and subclasses.
- Private: only the class.

Only private access modifer enforced (through name mangling).
It is however good practice to use protected.

The mode is defined through frontal underscores

- Public: no underscore.
- Protected: single underscore.
- Private: double underscore.

```
class BankAccount:
    id = 7312835182 # Public
    _name = "John Doe" # Protected
    __balance = 0   # Private
```

Getter

- Private attributes cannot be accessed outside the class and thus must be shared to a method known as a getter.

```
class BankAccount:
    def __init__(self, initial_balance):
        self.__balance = initial_balance
    def get_balance():
        return self.__balance
```

We can get attribute's value without being able to mutate.

- Can add logic to the getter.

Setter

- Private attributes cannot be mutated outside the class and thus must be updated through a method known as a setter.

```
class BankAccount:
    def __init__(self, initial_balance):
        self.__balance = initial_balance
    def set_balance(new_balance):
        if new_balance < 0:
            self.overdraft = True
            return
        self.__balance = new_balance
```

- Can add logic to the setter.

Let’s Practice

- Create a class **BankAccount** which has a constructor that takes in an initial balance. The balance should be updated only through two methods **deposit** and **withdraw** which both take in a number representing deposit or withdrawal amount. Add logic to guard against overdraft.

Modules and Packages

PCC 149-152, 173-179

Terminology

- A module is a Python file w/ reusable elements.
  - Classes and functions
- A package is a folder with at least 1 module and a special __init__.py file.
- You can nest packages into packages which become sub-packages.

Example


/
algorithms/
__init__.py
euclidean.py
math/
__init__.py
basic.py
main.py
screen.py
code directory
package
special init file
module
sub-package
special init file
module
entry point
not a module

Use init to expose

```
/
    algorithms/
        __init__.py
        euclidean.py
        math/
            __init__.py
            basic.py
    main.py
    screen.py
```

```python
# /algorithms/math/basic.py
import math

def add(x, y):
    """Add y to x"""
    return x + y
def sub(x, y):
    """Subtract y from x"""
    return x - y
def mul(x, y):
    """Multiply x by y"""
    return x * y
def div(x, y):
    """Divide x by y"""
    return x / y
def sqrt(x):
    """Square root x"""
    return math.sqrt(x)
```

```python
# /algorithms/math/__init__.py

from .basic import add, sub,
mul, div, sqrt
```

```python
# /algorithms/euclidean.py

from .math import mul, sqrt,
add


def hypotenuse(x, y):
    return sqrt(add(mul(x, x),
        mul(y, y)))
```

Import package as normal

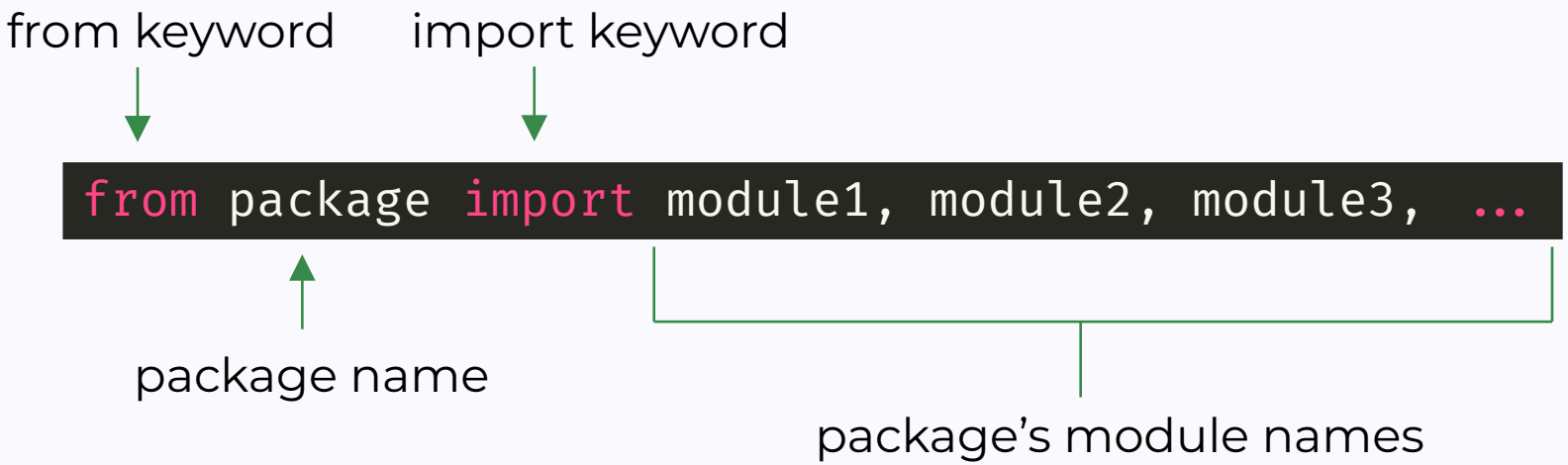
```
/
    algorithms/
        __init__.py
        euclidean.py
        math/
            __init__.py
            basic.py
    main.py
    screen.py
```

```python
# /main.py

from algorithms import
hypotenuse, add

print("The hypotenuse of
the 3, 4 triangle is: " +
str(hypotenuse(3, 4)))

print("1 + 1 = " +
str(add(1, 1)))
```

```python
# /algorithms/__init__.py

from .euclidean import
hypotenuse
from .math import add
```

```python
# /algorithms/euclidean.py

from .math import mul, sqrt,
add


def hypotenuse(x, y):
    return sqrt(add(mul(x, x),
        mul(y, y)))
```

We import with from import



Different packages

- Import python std packages via their name

```
from math
```

- Import external packages via their name

```
from numpy
```

  - Will need to install first with pip

```
pip install numpy
```

  - Can see all packages from pypi.org

```
from algorithms.math
```

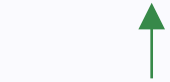
- Import user defined packages via absolute

```
from .math
```

- Import user defined packages via relative

I want to import everything

```
from package import *
```



* wildcard

Entry file special case

- The entry file is not loaded as a module and for that reason when you run it you cannot use relative imports as it is the start.
- Always use absolute paths.

Let’s practice

```
calculator_project/
├── main.py
└── calculator/
    ├── __init__.py
    ├── add.py
    ├── subtract.py
    ├── multiply.py
    └── divide.py
```

Design the following project on your computer where when **main** executes it asks the user to input a first number, second number, and finally the operation to perform.

Error Handling

PCC 192-199

Sometimes our code crashes

- When executing some code Python may not know what to do and throw an exception.
- If an exception is **not caught**, then Python crashes the runtime.

```
x = 0
y = 10 / x
```

```
ZeroDivisionError: division by zero
```

We protect our at risk with try-except.

- If an exception may be thrown it is best to surround it with a try-except block specifying the expected exception.

```
try:
    x = int(input("Enter a number: "))
    y = 10 / x
except ZeroDivisionError:
    print("Thou shall not giveth a 0")
```

Use else to set what happens on non exception.

- If we have specific code we only want to run if the try block did not produce an error, we can use an else block.

```
try:
    x = int(input("Enter a number: "))
    y = 10 / x
except ZeroDivisionError:
    print("Thou shall not giveth a 0")
else:
    print(f"Your code is {y}")
```

You can fail silently if you want

- Sometimes you do not want to inform the user that an error was produced, so you can supply pass to the except block.

```
try:
    x = int(input("Enter a number: "))
    y = 10 / x
except ZeroDivisionError:
    pass
else:
    print(f"Your code is {y}")
```

Use exception if unsure type

- If you are unsure of the type, or have potentially multiple exception types that you want to act similarly, use Exception as type.

```
try:
    x = int(input("Enter a number: "))
    y = 10 / x
except Exception:
    pass
else:
    print(f"Your code is {y}")
```

Let’s practice

- Write a program that will safely divide two user provided numbers, and will continuously prompt until success.

Debugging

We make mistakes

- In fact the first time you write code there will be problems.
- These bugs, we need a way to hunt them down.
- Debugging!

2 Main Techniques

1) Validating intermediate variables states with printing.

2) Using purpose built debugger.

Printing

- Sometimes printing the state of variables at a given moment can help spot bugs.

```
def find_largest(numbers):
    largest = 0
    for num in numbers:
        if num > largest:
            largest = num
    return largest

# Test case
print(find_largest([-10, -5, -20]))
```

0

Adding print statements

```python
def find_largest(numbers):
    largest = 0
    print("Initial largest:", largest)
    for num in numbers:
        print("Checking number:", num)
        if num > largest:
            print(f"{num} is greater
             than {largest} → updating largest")
            largest = num
        else:
            print(f"{num} is not greater than {largest}")
    return largest

# Test case
print("Result:", find_largest([-10, -5, -20]))
```

```
Initial largest: 0
Checking number: -10
-10 is not greater
than 0
Checking number: -5
-5 is not greater
than 0
Checking number: -20
-20 is not greater
than 0
Result: 0
```

Using Python pdb

```python
def find_largest(numbers):
    largest = 0
    for num in numbers:
        breakpoint()
        if num > largest:
            largest = num
    return largest


# Test case
print(find_largest([-10, -5, -20]))
```

```
python main.py
```

```
→ breakpoint()
(Pdb) p largest
0
(Pdb) p num
-10
(Pdb) p num > largest
False
(Pdb) n
→ if num > largest:
(Pdb) n
→ for num in numbers:
(Pdb) p num
-10
(Pdb) n
→ breakpoint()
(Pdb) p num
-5
(Pdb) n
→ if num > largest:
(Pdb) n
→ for num in numbers:
(Pdb) c
→ breakpoint()
(Pdb) p num
-20
(Pdb) c
0
```

Common debugger commands

- n (next): go to next line of current function.
- s (step): go to very next line (if function call → go inside).
- p *expression*: prints the expression (vars included).
- c (continue): go to the next breakpoint.
- And so much more you can use from the docs.
  - https://docs.python.org/3/library/pdb.html

Let’s practice

- Try to solve the bug in the following code which you can find on canvas.

```
def find_max(nums):
    max_val = nums[0]
    for i in range(1, len(nums) - 1):
        if nums[i] > max_val:
            max_val = nums[i]
    return max_val
```

And some more

```
def repeat_message(message, times):
    return message * times

msg = input("Enter a message: ")
count = input("How many times? ")
print(repeat_message(msg, count))
```

```
def sum_even(numbers):
    total = 0
    for num in numbers:
        if num % 2 == 1:
            total += num
    return total

print("Sum of even numbers:", sum_even([1, 2, 3, 4, 5, 6]))
```