```python
# Tutorial Week 10
# OOP Python
# __str__, try...except revision, Abstract classes and
# Polymorphism
# author: B. Schoen-Phelan
# date: Nov 2020

# some try...except revision
# it's great for input validation!
# this version is cleaned up and has more comments
# compared to the life demo in the tutorial

# version 1: just some input validation
term = int(input("Input please:"))
# run this and enter a character and see what happens
#
print(term)
print("rest of program")

# version 2: do a try except
try:
   term = int(input("Input please:"))
   # enter a character, see what happens
except:
   pass
#
# print(term)
print("rest of program")

# version 3: print something useful in except
try:
   term = int(input("Input please:"))
   # enter a character, see what happens
except Exception as e:
   print(e)

# print(term)
print("rest of program")
```

```
# version 4: print something useful but also consider
# that you might not have been able to think about
# every eventuality that could go wrong
try:
   term = int(input("Input please:"))
   # enter a character, see what happens
except ValueError as ve:
   print("A value error has happened")
   print(ve)
   term = 0 # keep any future usage of term from
            # crashing the program if we ran into this
except Exception as e:
   print("I didn't realise this could happen")
   print(e)
   term = 0
#
print(term)
print("rest of program")

# raising errors to keep your functions save to use
def my_function(value):
   if value > 5:    # often used for business logic
       raise ValueError("this is not allowed")

# version 1: the crash
my_function(7) # this will crash

# version 2: the save exit
try:
   my_function(7)
except ValueError as ve:
   print(ve)
except Exception as e:
   print("This was unexpected")
   print(e)

# version 3: running into unexpected issues
# or forcing them (in this case)
def my_function(value):
    if value > 5:     # often used for business logic
        raise ValueError("this is not allowed")

    value/0  # this will force an exception because div
by 0 is not allowed


try:
    my_function(3) # different value so we don't run
into the issue
except ValueError as ve:
    print(ve)
except Exception as e:
    print("This was unexpected")
    print(e)


# back to the Salary and Employee example from last
# week
# how done with "private" variables
# and how represent an object not with a memory address
# if your method only gets or sets a variable: it
# should
# absolutely be a @property!
class Salary:

    def __init__(self, pay, bonus):
        self.__pay = pay
        self.__bonus = bonus

    # if you want to control the string representation
    # of an object
    # rather than returning something like
    # <__main__.Salary object at 0x7fe97e906100>
    # similar but slightly different is __repr__
    def __str__(self):
        return f"I earn {self.pay_prop} and my bonus is:
{self.bonus_prop}"
```

```
    @property
    def pay_prop(self):
        return self.__pay

    @pay_prop.setter
    def pay_prop(self, value):
        self.__pay = value

    @property
    def bonus_prop(self):
        return self.__bonus

    @bonus_prop.setter
    def bonus_prop(self, value):
        self.__bonus = value

    # returns a calculation
    def annual_salary(self):
        return (self.pay_prop * 12) + self.bonus_prop


# composition:

class Employee:
    def __init__(self, name, age, pay, bonus):
        self.__name = name
        self.__age = age
        self.__salary_object = Salary(pay, bonus)

    @property
    def age_prop(self):
        return self.__age

    @age_prop.setter
    def age_prop(self, value):
        self.__age = value

    @property
    def name_prop(self):
        return self.__name

    # if you don't provide a setter property, then this
    # variable cannot be set!

    @property
    def salary_prop(self):
        return self.__salary_object

    def total_salary(self):
        return self.salary_prop.annual_salary()


anna = Employee("Anna", 25, 2500, 10000)
# print(anna.total_salary())
print(anna.salary_prop)     # gives the object memory
                            # location without str


class DifferentMethodsClass:
    class_attribute = "This is a class attribute"

    def __init__(self):
        self.instance_attributes = "This is an instance
attribute"

    def instance_method(self):   # usual argument self
        print('instance method called', self)

    @classmethod
    def class_method(cls):   # notice what's new in the
                             #argument list
        print('class method called', cls)
        # print(self.instance_attributes)   # this will
                                            # fail
        print("class attribute: ", cls.class_attribute)

    @staticmethod
    def static_method():   # notice nothing in the
```

```
                          # argument list
        print('static method called')

# causes an error:
print("instance attribute",
DifferentMethodsClass.instance_attributes)


demo = DifferentMethodsClass()
# all of these work just ine
demo.instance_method()
demo.class_method()   # does not have access to instance
                      # variables
demo.static_method()
#

DifferentMethodsClass.class_method()   # works fine
DifferentMethodsClass.static_method()   # works fine
# DifferentMethodsClass.instance_method()   # error

# difference between class attribute and
# instance attribute: class attribute, if mutable
# can be changed:
class A:
    CLASS_ATTRIBUTE = ["a", "b", "c"]

    def __init__(self):
        self.instance_attribute = ["x", "y"]

    def instance_method(self):
        print("instance method called")

    @classmethod
    def class_method(cls):
        print("class method called")

    @staticmethod
    def static_method():
        print("static method called")
```

```
demo1 = A()
demo2 = A()
print(demo1.CLASS_ATTRIBUTE)
print(demo2.CLASS_ATTRIBUTE)
demo2.CLASS_ATTRIBUTE.append("x")
print(demo1.CLASS_ATTRIBUTE) # demo1 can read that
                             # change


print(demo1.instance_attribute)
demo1.instance_attribute.append("a")
print(demo1.instance_attribute)
print(demo2.instance_attribute)
# demo2 cannot read that change, as it has its own
# version of the instance attribute


# example adapted from Real Python
#
#https://realpython.com/instance-class-and-static-metho
ds-demystified/
import math


class Pizza:
   def __init__(self, ingredients, pizza_size=3):
       self.ingredients = ingredients
       self.pizza_size = pizza_size
       print("in init")

   def __str__(self):
       return f'Pizza({self.ingredients}) of size:
{self.pizza_size}'

   # add some class methods
   @classmethod
   def margherita(cls):
        return cls(['mozzarella', 'tomatoes'])

    @classmethod
    def prosciutto(cls):
        return cls(['mozzarella', 'tomatoes', 'ham'])

    @staticmethod
    def pizza_area(r):
        return r ** 2 * math.pi


pizza = Pizza(['cheese', 'tomatoes'])
print(pizza)
print(Pizza.prosciutto())
print(Pizza.margherita())

# print(Pizza.ingredients) # causes an error, cannot
# access instance variables
print(Pizza.pizza_area(3))

# example modified from
#
https://www.geeksforgeeks.org/abstract-classes-in-pytho
n/#:~:text=An%20abstract%20class%20can%20be,is%20called
%20an%20abstract%20class.
from abc import ABC, abstractmethod


# this is NOT an abstract base class, although
# derived from ABC

class Polygon:

    # abstract method
    def no_of_sides(self):
        pass


class Triangle(Polygon):  # this allows no
```

```python
# implementation, hence not a real abstract class
    pass

t = Triangle()




# now with the decorator and an implementation. If
# you don't define the implementation in the child
# class you will get an error

class Polygon(ABC):

    @abstractmethod
    def no_of_sides(self):
        pass


class Triangle(Polygon):
    # overriding abstract method
    def no_of_sides(self):
        print("I have 3 sides")
        # pass


t = Triangle()
t.no_of_sides()

# add another class
class Pentagon(Polygon):

    # overriding abstract method
    def no_of_sides(self):
        print("I have 5 sides")
    # pass  #causes an error. Also child of child of
    # abstract needs to provide implementation

p = Pentagon()
```

```
p.no_of_sides()






# abstract classes may contain a mixture of abstract
# and normal methods

class Polygon(ABC):

   @abstractmethod
   def no_of_sides(self):
       pass

   def what_am_I(self):
       print("I am a parent Polygon")


class Triangle(Polygon):
   # overriding abstract method
   def no_of_sides(self):
       print("I have 3 sides")


t = Triangle()
t.no_of_sides()
t.what_am_I()


# or the same with an overriden method and a call to
# super
class Polygon(ABC):

   @abstractmethod
   def no_of_sides(self):
       pass
```

```python
    def what_am_I(self):
        print("I am a parent Polygon")




class Triangle(Polygon):
    # overriding abstract method
    def no_of_sides(self):
        print("I have 3 sides")

    # pass

    def what_am_I(self):
        print("I am a Triangle child class")
        super().what_am_I()


t = Triangle()
t.no_of_sides()
t.what_am_I()  # everything works as expected

# abstract classes can have abstract methods and
# abstract properties

class Polygon(ABC):

    @abstractmethod
    def no_of_sides(self):
        pass

    def what_am_I(self):
        print("I am a parent Polygon")

    @property
    @abstractmethod # the abstract decorator should be
                    # the last one in the list
    def length(self):
        pass




class Triangle(Polygon):

    def __init__(self):
        self.__x = 0

    def no_of_sides(self):
        print("I have 3 sides")

    def what_am_I(self):
        print("I am a Triangle child class")
        super().what_am_I()

    @property
    def length(self):
        return self.__x

    @length.setter
    def length(self, value):
        self.__x = value


# print(Triangle.__mro__)
t = Triangle()
t.no_of_sides()
print(t.length)
t.length = 5
print(t.length)

# polymorphism

# an example that you know already
```

```python
my_list = ["a", "b", "c"]
my_string = "Hello World"

print(len(my_string))
print(len(my_list))


# same method name in the classes
class Dog:
    def speak(self):
        print("wouff wouff")

class Cat:
    def speak(self):
        print("meow")


def let_animals_speak(a):
    a.speak()

d = Dog()
c = Cat()

# calls the correct method depending on the
# type
let_animals_speak(d)
```