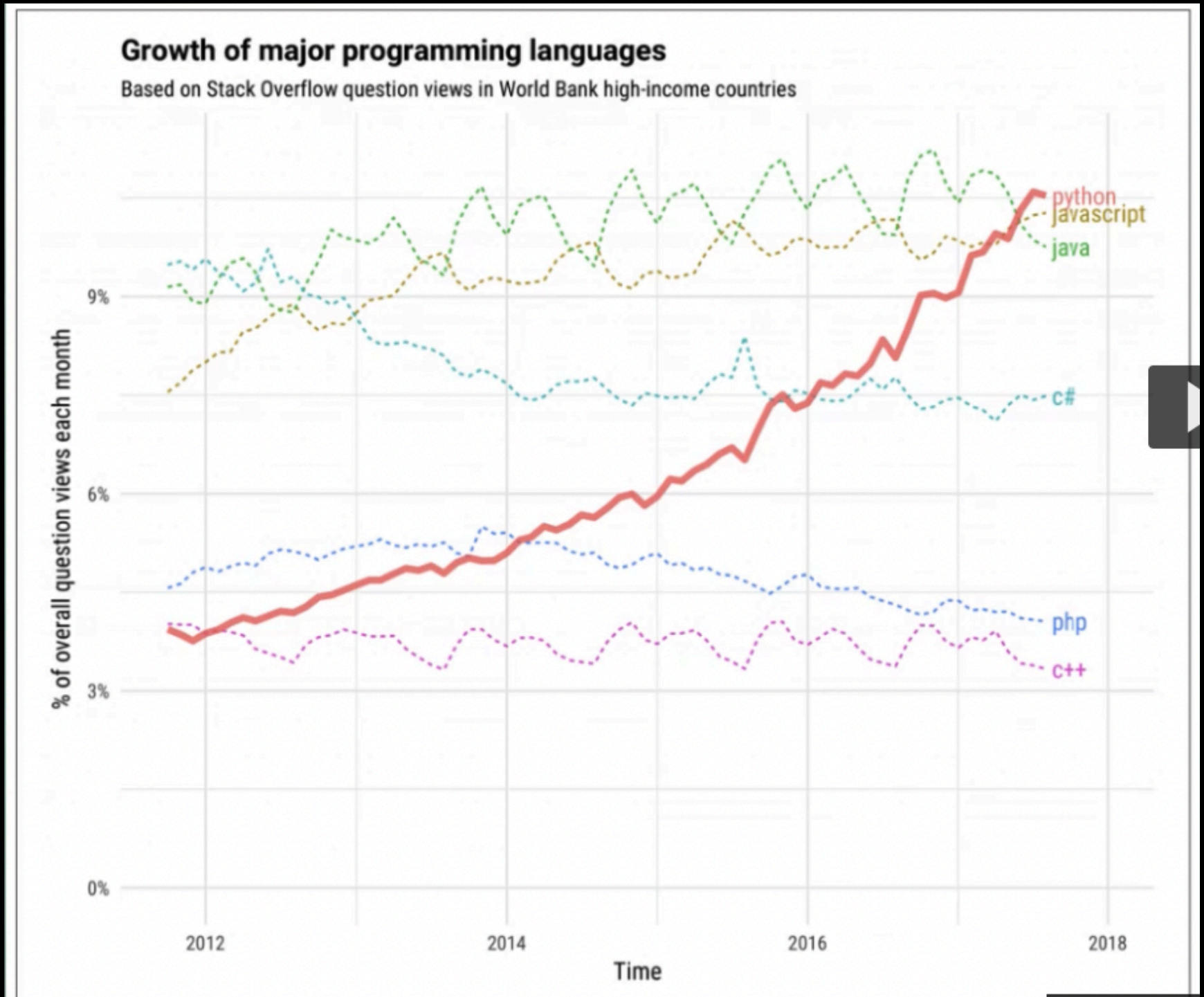
Include all that apply in your communication:

For subject content related questions:
Course number
Module number
Video title
Quiz or Exam name/number; Question number (or description of specific question)
Screenshots/links to support your question
Details of your attempt to understand/troubleshoot on your own

For non-subject content related questions:
Course(s) you are enrolled in
Screenshots/links to support your question
Details of your attempt to understand/troubleshoot on your own

Fastest growing programming language.


Growth of major programming languages
Based on Stack Overflow question views in World Bank high-income countries
% of overall question views each month
9%
6%
3%
0%
2012
2014
2016
2018
Time
python
javascript
java
c#
php
c++

1. Easy to Read
2. Python is free.
3. People enjoy coding in Python

Python is slower.
Python uses of white space.

Module 1:

Data are objects
1. Type
2. Unique ID
3. Value
4. Reference Count

Basic Data Types			
Name	Type	Mutable?	Examples
Boolean	`bool`	no	`True, False`
Integer	`int`	no	`47, 25000, 25_000`
Floating point	`float`	no	`3.14, 2.7e5`
Complex	`complex`	no	`3j, 5 + 9j`
String	`str`	no	`'alas', "alack", '''a verse attack'''`
List	`list`	yes	`['Winken', 'Blinken', 'Nod']`
Tuple	`tuple`	no	`(2, 4, 8)`
Bytes	`bytes`	no	`b'ab\xff`
ByteArray	`bytearray`	yes	`bytearray(...)`
Set	`set`	yes	`set([3, 5, 7])`
Frozen set	`frozenset`	no	`frozenset(['Elsa', 'Otto']`

Mutable and Immutable

Strongly type. (Can't change int)

1. Difference between literals and variables.
2. What variable names can and cannot begin with and what they can and cannot contain.
3. What reserved words are used in Python.
4. How to assign values.

1. Python is dynamically typed. (this means variables in Python are just names.)
   a = 7, A is a name tag on object 7
   The name of the variable is a reference to an object.
   In C you have declare both variable and object

booleans (true / false)

bool("") -> False
bool(0) -> False
bool(False) -> False
bool(false) -> error

vowels = 'aeiou'
letter = 'o'
letter in vowels -> True

Integer division

9 / 5 -> 1.8

9 // 5 -> 1

15 / 4 -> 3.75

15 // 4 -> 3

Modulus

9 % 5 -> (what's left over 4)

15 % 4 -> 3

Bases

binary number
0b0001 is same as 0b1

bin(56) -> 0b111000

Type

```
x = 1
type(x)
int

x = bool(x)
type(x)
bool

x = 1.74
type(x)
float

x = int(x)
```

```
type(x)
int
x -> 1

upcasting
1 + 1.7
2.7 (upcat to float when add integer and float)

y = '7'
type(y)
str
int(y) -> 7 (It's okay to convert number string into integer)
z = '7.7'
folat(z) -> 7.7

True + 2 -> 3
False + 2 -> 2

Tuples:

name = 'Mike'
type(name)
<class 'str'>

name = 'Mike',
type(name)
<class 'tuple'>
name -> ('Mike',)

names = 'Mike', 'Jenny', 'Alan', 'Alec'
type(names)
<class 'tuple'>
```

```
names -> ('Mike', 'Jenny', 'Alan', 'Alec')
a, b, c, d = names
a -> 'Mike'
b -> 'Jenny'
c -> 'Alan'
d -> 'Alec'

type(a)
<class 'str'>

Tuples:
namelist = ['Mike', 'Jenny', 'Alec', 'Alan']
tuple(namelist) -> ('Mike', 'Jenny', 'Alec', 'Alan')
('Mike',) + ('Jenny',) -> ('Mike' 'Jenny')
('Mike',) * 3 -> ('Mike', 'Mike', 'Mike')
'Mike', * 3 -> Error

t1 = ('Hello')
t2 = ('world')
id(t1)

t1 += t2
t1 -> ('Hello', 'world')
id(t1) -> it's different because it's now pointing to a new tuple.

List:
name = 'Mike Yom'
name.split()
['Mike', 'Yom']
birthday = '9/10/67'
birthday.split() -> ['9/10/67'] (Because no spaces)
birthday.split('/') -> ['9', '10', '67']
```

```
names = ['Mike', 'Jenny', 'Alan']
names.insert(2, 'Alec') -> ['Mike', 'Jenny', 'Alec', 'Alan']
del names[0] -> ['Jenny', 'Alec', 'Alan']
names.remove('Alec') -> ['Jenny', 'Alan']
'Jenny' in names -> True
'Mike' in names -> False
names.count('Jenny') -> 1
mylist = [1, 1, 1, 1, 1, 3, 4, 6]
mylist.count(1) -> 5 ( 5 ones)


names = ['Mike', 'Jenny', 'Alec', 'Alan']
for i in names:
   print(i)

ages = [54, 46, 17, 15]

for i, j in zip(names, ages):
    print(i, j)
Mike 54
Jenny 46
Alec 17
Alan 15

num_list = []
for i in range(1, 6):
   num_list.append(i)

num_list
[1, 2, 3, 4, 5, 6]
```

List Comprehension
```
from math import pi
[str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Copy and Deep Copy:
```
a -> [1, 2, 3]
b = a.copy()
c = list(a)
d = a[:]
all have now [1, 2, 3]
a[0] = 56 -> [56, 2, 3]

e -> [1, 2, [8, 9]] (8 and 9 are list mutable)

f = e.copy()
f -> [1, 2, [8, 9]]

e[2][0] = 56
e -> [1, 2, [56. 9]]
f -> [1, 2, [56. 9]]

e[0] = 11
e -> [11, 2, [56. 9]]
f -> [1, 2, [56. 9]]
(Because lists are mutable and tuple are not mutable)

Deep copy
e = [11, 2, [56. 9]]
import copy
g = copy.deepcopy(e)
```

```
g -> [11, 2, [56. 9]]
e[2][0] = 99
e -> [11, 2, [99. 9]]
f -> [11, 2, [56. 9]] (that did not apply reference) (copy keep references)

Dictionaries:

Key Value Pairing
Order doesn't matter

tup_ex -> (['a', 'b'], ['c', 'd'])
list_ex -> ['ab', 'cd']
dict(tup_ex) - {'a': 'b', 'c': 'd'}
dict(list_ex) -> {'a': 'b', 'c': 'd'}
x = dict(tup_ex)
x -> {'a': 'b', 'c': 'd'}
x['a'] = 'j'
x -> {'a': 'j', 'c': 'd'}

ale.get('Yankees')
ale.get('Yankees', 'Not a key') -> 'Not a key'
ale.get('New York') -> Yankees

Difference between return and print
type(return_calculate(3, 5)) -> int
type(print_calculate(3, 5)) -> NoneType

1 + \
1

(1 +
```

```
1
) -> 2
```

Module 2

```
Formatting strings

Combination
Duplication
Indexing
Slicing
Length
Strip
Replace

Multi length string
x = ```
hello
world
it's
me
```
print(x)

'hello\nworld' -> 'hello\nworld'
y = `hello\nworld'
print(y)
hello
world
```
```

```
"\"I did nothing!\" he said" -> '"I did nothing!" he said'

x = "Here is a backslash: \\"
print(x) -> Here is a backslash: \

raw string and printed string
```

Strip Case Alignment

```
world = '   earth   '
print(world) -> '   earth   '
world.strip() -> 'earth'

setup = 'a duck goes into a bar…'
setup.strip('.') -> 'a duck goes into a bar'
setup.capitalize() -> 'A duck goes into a bar' (First word)
setup.title() -> 'A Duck Goes Into A Bar'
setup.upper()
setup.lower()
setup.swapcase()

setup.center(30) (center in 30 spaces)
setup.ljust(30) (left justified in 30 spaces)
setup.rjust(30) (right justified in 30 spaces)
```

Old Style Formatting

```
Old Style Python 2+
New Style Python 2.6+
f-string Python 3.6+
```

```
Old style
team = 'Yankees'
"My favorite team is the %s" % team
'My favorite team is the Yankees'

print('%s' % 42)
print('%d' % 42)
print('%x' % 42)
42
42
2a

print('%s' % 7.03)
print('%f' % 7.03)
print('%e' % 7.03)
7.03
7.030000
7.030000e+00

print('%d%%' % 100)
100%

city = 'New York'
'My favorite team is the %s %s' % (1998, team)
'My favorite team is the 1998 Yankees'

'My favorite team is the %s %s' % (city, team)
'My favorite team is the New York Yankees'

print('%s' % team)
print('%12s' % team) (12 spaces shift to right)
print('%s12' % team) (put 12 after s)
```

```
print('%12.3s' % team) (12 spaces 3 char right align)
print('%-12.3s' % team) (left align take 3 character)
Yankees
     Yankees
Yankees12
          Yan
Yan
```

New Style

```
team = "Yankees"
'{}'.format(team) -> 'Yankees'
'My favorite team is the {1} {0}.format(team, city)
'My favorite team is the New York Yankees'

'My favorite team is the {city} {team}'.format(city = 'New York', team = 'Yankees')
'My favorite team is the New York Yankees'

print('My favorite team is the {} {}'.format(city, team))
print('My favorite team is the {:10s} {}'.format(city, team))
print('My favorite team is the {:>10s} {:^10s}'.format(city, team.upper()))(center)
print('My favorite team is the (:>10s} {:^10.4s}'.format(city, team)) (max char of 4)
My favorite team is the New York Yankees
My favorite team is the New York    Yankees (10 spaces)
My favorite team is the   New York  YANKEES
My favorite team is the   New York    Yank
```

F-Strings

```
f'My favorite team is the {city} {team}'
'My favorite team is the New York Yankees'
```

```
f'My favorite team is the {city} {team.upper()}'
'My favorite team is the New York YANKEES'

print('My favorite team is the {} {}'.format(city, team))
print(f'My favorite team is the {city} {team}')

print('My favorite team is the {:10s} {}'.format(city, team))
print(f'My favorite team is the {city:10s} {team}')

print('My favorite team is the {:>10s} {:^10s}'.format(city, team.upper()))(center)
print(f'My favorite team is the {city:10s} {team.upper():^10s}') team.upper()))(center)

print('My favorite team is the (:>10s} {:^10.4s}'.format(city, team)) (max char of 4)
print(f'My favorite team is the (city:10s} {team:^10.4s}') (max char of 4)
```

Control Structure

```
disaster = True
if disaster:
   print('AAAAHHHHH')
else:
   print(;WHEEEE')

color = "mauve"

if color == "red":
     print("It's a tomato")
elif color == "green":
     print("It's a green pepper")
elif color == "bee purple":
     print("I don't know what it is, \
```

```
     but only bees can see it")
else:
     print("I've never heard of the color", color)
```

Walrus Operator

```
:=

tweet_limit = 280
tweet_string = "Blah" * 50
diff = tweet_limit - len(tweet_string)

if diff >= 0:
     print("A fitting tweet")
else:
     print("Went over by", abs(diff))

################

tweet_limit = 280
tweet_string = "Blah" * 50
diff = tweet_limit - len(tweet_string)

if diff := tweet_limit - len(tweet_string) >= 0: #  operator
     print("A fitting tweet")
else:
     print("Went over by", abs(diff))

################

temp = float(input('Please input the temperature of interest: '))
```

Loops

```
family = ['Mike', 'Jenny', 'Alec', 'Alan']

for i in family:
  print(i)

while a <= 3:
  print(a)
  a += 1
```

Breaks

```
stops loop
family = ['Mike', 'Jenny', 'Alec', 'Alan']
classes = ['DTSC520', 'DTSC550', 'DTSC600', 'DTSC660']

for i in family:
  print(i)
  if i == 'Jenny':
     break


for i in family:
     print(i)
     for j in classes:
          print(j)
          if j == 'DTSC600':
               break
```

breaking two loops

```
for i in family:
     print(i)
     for j in classes:
          print(j)
          if j == 'DTSC600':
               break
     else:
          continue
     break
```

continue is pass over current iteration

```
family = ['Mike', 'Jenny', 'Alec', 'Alan']

for i in family:
     if i == 'Alec':
          continue
     print(i)
```

```
while True:
     value = input('Integer, please [q to quit]: ')

     if value == 'q':
          break

     number == int(value)
     if number % 2 == 0:
          continue
```

```
     print(number, "squared is", number * number)
```

Range

```
for i in range(0, 3):
     print(i)

for i in range(0, 10, 2):
     print(i)


for i in range(2, -1, -1):
     print(i)


def acronym():
     target = input('What phrase do you want? ')

     ac = ""
     for i in target.split():
          ac = ac + i[0]
     print(ac.upper())
```

Module 3

```
print(f'Hello {sys.argv[1]}!'

####################################

import sys
```

```
print(sys.argv[0])
print(sys.argv[1])

x = int(sys.argv[2])

def example():
     print(f'Hello {sys.argv[1]}!')
     print(x**2)

example()

python3 01_example.py Mike 5

####################################
```

sys

```
import sys

fam = {'Jenny': 'Mother',
           'Mike': 'Father',
           'Alec': 'Son',
           'Alan': 'Son'}

name = sys.argv[1]

def relation():
     print(name, "is the", fam[name])

relation()
```

```
python3 02_name.py Mike
```

Functions Overview

```
python3

exec(open('hello.py').read())

Function Review
```

```
def echo(anything):
     return anything + ' ' + anything

echo('hello')
```

```
import sys


def commentary(color):
     if color == 'red':
          return "It's a tamato"

     elif color == 'green':
          return "It's a green pepper"
```

```
     elif color == 'bee purple':
          return "I don't know what it is, but only bee can see it"

     else:
          return f"I've never heard of the color {color}."


my_color=str(sys.argv[1])

print(commentary(my_color))

python3 color.py red
```

```
default

def menu2(wine, entree, dessert = 'cake'):
     return{'wine': wine, 'entree': entree. 'dessert': dessert}

menu2('riesling', 'hot dog')
{'wine': 'riesling', 'entree': 'hot dog', 'dessert':'cake'}

menu2(('riesling', 'hot dog', 'cookies')
{'wine': 'riesling', 'entree': 'hot dog', 'dessert':'cookies'}
```

Exploding and Gathering

```
def print_args(*args):
     print(args)
```

```
print_args(1, 3, 4, 2, 6, 4.25, 'hello')
(1, 3, 4, 2, 6, 4.25, 'hello')

required argument

def print_args(req1, req2, * args):
     print('Required:', req1)
     print('Required also:', req2)
     print('All the rest', args)

Required: 4
Required alos: 3
All the rest: (5, 2, 3, 134.243, 'hello')
```

Arguments passed to a function in the same order that the function's parameters are listed are called (positional) arguments

positional arguments
keyword arguments

```
dictionary
def print_kwargs(**kwargs):
     print(kwargs)

print_kwargs(wine = 'riesling', entree = 'chicken', dessert = 'cookies')
#print in dictionary
```

```
{'wine': 'riesling', 'entree': 'chicken', 'dessert':'cookies'}
```

Keyword and Mutable Argument

```
def print_date(data, *, start, end):
     for i in (data[start:end]):
          print(i)

data
['a', 'b', 'c', 'd', 'e', 'f']

print_data(data, start = 2, end = 4)
c
d
```

```
outside = ['one', 'fine', 'day']

def mangle(arg):
     arg[1] = 'terrible'

mangle(outside)
outside
['one', 'terrible', 'day']
```

Docstrings

```
def addup(x, y):
    '''add arguments and return sum.'''
     return x + y

# ipython
# import addup
# addup.addup(2, 5)
# help(addup)
# from numpy import array
# help(array)
```

Lambda

```
def addup(x, y):
    return x + y

outcome = addup(3, 4)

print(outcome)

func = lambda x, y: x + y

outcome = func(3, 4)
```

```
def enliven(word):
    return word.capitalize() + '!'

def edit_story(words, func):
    for word in words:
        print(func(word))

stairs = ['thud', 'meoq', 'thud', 'hiss']

edit_story(stairs, enliven)

edit_story(stairs, lambda word: word.capitalize() + '!')

names = ['Gregory S. Longo', 'Mike Morabito', 'Javier Leon', 'Ashley Wiley']

names.sort(key = lambda x: x.split(" ")[-1].lower())
```

Decorators

```python
def document_it(func):
    def new_function(*args, **kwargs):
        print('Running function:', func.__name__)
        print('Positional arguments:', args)
        print('Keyword arguments:', kwargs)
        result = func(*args, **kwargs)
        print('Result', result)
        return result
    return new_function
```

```python
def add_ints(a, b):
    return a + b
```

```python
add_ints(3, 5)
```

```
8
```

```python
decorated_add = document_it(add_ints)
decorated_add(3, 5)
```

```
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result 8

8
```

```python
@document_it
def add_ints(a, b):
    return a + b
```

```python
def square_it(func):
    def new_function(*args, **kwargs):
        result = func(*args, **kwargs)
        return result * result
    return new_function
```

```python
@document_it
@square_it
def add_ints3(a, b):
    return a + b
```

```python
add_ints3(3, 5)
```

```
Running function: new_function
Positional arguments: (3, 5)
Keyword arguments: {}
Result 64

64
```

```python
def test(a, b):
    return a - b
```

Exceptions except

```
short_list = [1, 2, 3]
position = 5
```

```
try:
    short_list[position]
except:
    print('Need a position between 0 and',
          len(short_list)-1, ' but got', position)
```

```
Need a position between 0 and 2  but got 5
```

```
# this is a "catchall" exception it works for all
# but they can be more specific
```

```
while True:
    value = input('Position [q to quit]? ')
    if value == 'q':
        break
    try:
        position = int(value)
        print(short_list[position])
    except IndexError as err:
        print('Bad index:', position)
    except Exception as other:
        print('Something else broke:', other)
```

```
Position [q to quit]? 1
2
Position [q to quit]? 3
Bad index: 3
Position [q to quit]? t
Something else broke: invalid literal for int() with base 10: 't'

Position [q to quit]?
```

when used inside a function with a parameter, an asterisk groups a variable number of positional arguments into a single _____ of parameter values (list)

Function Test.

- def cube(x) -> missing
- Yoda!!!
- triangle(a=4, b=8, c=11)
- We can explode/gather keyword arguments with * (false)
- None == {} -> False
- help(triangle)
- positional arguments
- def time(hours = 24):
- explode arguments -> *args

- def me(name, hometown): -> don't forget :

- When used inside a function with a parameter, an asterisk groups a variable number of positional arguments into a single ___ of parameter values (args)

- def hello():
      '''Returns a greeting''' is called a (maybe docstring)

- @upper
  def hello(name):
      print(f'Hello, {name}!')
  in this code, @upper is a (decorator)

- You could pass a keyword argument that has the same name as a positional parameter. Because of that, you can use what type of arguments to get around that? (keyword only arguments)

- def triangle(a , b, c):
  knowing nothing else, how would you run the function using keyword arguments if the values of a, b and c are 4, 8, 11 respectively?

- positional

- None

- What character do you use to explode arguments? (*args)

- What is code that is executed when an associated error occurs? (except)

- print(f'My classes are: {args}')

- A decorator is a function that takes one function as input and returns another function.

Object-Oriented Programming

everything in python is an object.

What are objects?
Objects are a type of data structure that contain both data and code.

Data: Attributes
Code: Methods

You can do differently to values.

7 = object

Create a new types of object

Classes: A mode that creates those boxes.

Classes are a blueprint to create these boxes.

Classes and Attributes

```
class Cat():
    pass

a_cat = Cat()
```

```
another_cat = Cat()

a_cat
```

Attributes

```
a_cat.age = 3
a_cat.name = 'Mr. Fuzzybuttons'
a_cat.nameis = another_cat

print(a_cat.age)
print(a_cat.name)
print(a_cat.namesis)
3
Mr. Fuzzybuttons
<__main__.Cat object at 0x7f8770f089a0>

a_cat.namesis.name = 'Mr. Bigglesworth'
a_cat.namesis.name
'Mr. Bigglesworth'
```

Methods

```
# Assign attributes with by initializing with __init__():
# Yes, these are two underscores, aka 'dunder'

class Cat():
    def __init__(self):
        pass

class Cat():
```

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

furball = Cat('Grumpy')

print(furball.name)
Grumpy
```

Inheritance

```
# Assign attributes with by initializing with __init__():
# Yes, these are two underscores, aka 'dunder'

class Car():
    pass

# child class, create Hyundai subclass from parent class

class Hyundai(Car):
    pass

issubclass(Hyundai, Car)
True

issubclass(Car, Hyundai)
False

class Car():
    def exclaim(self):
        print("I'm a Car!")

class Hyundai(Car):
```

```
    pass

give_me_a_car = Car()
give_me_a_huyndai = Hyundai()
give_me_a_car.exclaim()
I'm a Car!

give_me_a_huyndai.exclaim()
I'm a Car!
```

Overriding and Adding Methods

```
class Car():
    def exclaim(self):
        print("I'm a Car!")

class Hyundai(Car):
    def exclaim(self):
        print("I'm a Hyundai")

give_me_a_car = Car()
give_me_a_huyndai = Hyundai()
give_me_a_car.exclaim()
I'm a Car!

give_me_a_huyndai.exclaim()
I'm a Huyndai

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

```
class MDPerson():
    def __init__(self, name):
        self.name = "Doctor " + name

class JDPerson():
    def __init__(self, name):
        self.name = name + ", Esquire"


person = Person('Fudd')
doctor = MDPerson('Fudd')
lawyer = JDPerson('Fudd')

print(person.name + "\n" + doctor.name + "\n" + lawyer.name)
```

```
Fudd
Doctor Fudd
Fudd, Esquire
```

Adding a method

```
class Car():
    def exclaim(self):
        print("I'm a Car!")

class Hyundai(Car):
    def exclaim(self):
        print("I'm a Hyundai")
    def need_a_push(self):
        print("A little help here?")
```

```
my_car = Car()
my_hyundai = Hyundai()

my_car.exclaim()
I'm a Car!

my_hyundai.exclaim()
I'm a Hyundai

my_hyundai.need_a_push()
A little help here?
```

Attribute access

Getters and setters

```
class Duck():
    def __init__(self, input_name):
        self.hidden_name = input_name
    def get_name(self):
        print('inside the getter')
        return self.hidden_name
    def set_name(self, input_name):
        print('inside the setter')
        self.hidden_name = input_name

don = Duck('Donald')
don.get_name()

don.set_name('Donna')
```

Object Oriented Programming

1. Inheritance (like a car class)(parent / child)
2. Encapsulation (data containing a function)
3. Abstraction (we removed unnecessary details, allowing the user to focus only on what is necessary) (when we create an object, there's a whole lot behind the scenes)
4. Polymorphism (what an object does when these is a method call depends on the class of the object) (have a function that works across different classes)

An object is simply a collection of data (variables) and methods (functions) that act on those data. Similarly, a class is a blueprint for that object.

```
########################################

Inheritance is the mechanism for creating a child class that can inherit behavior and
properties from a parent(derived) class.

class Animal:

    def __init__(self, name):
        self.name = name
        print(self.name + " was adopted.")

    def run(self):
        print("running!")


class Dog(Animal):
```

```
    def __init__(self):
        super().init

    def bark(self):
        print("woof!")


# new dog behavior inherited from Animal parent class
spot = Dog("spot") #=> spot was adopted.
spot.run() #=> running!

########################################

Encapsulation is the method of keeping all the state, variables, and methods private unless
declared to be public.

class Fish:

    def __init__(self):
        self.__size = "big"

    def get_size(self):
        print("I'm a " + self.__size + " fish")

    def set_size(self, new_size):
        self.__size = new_size

# using the getter method
oscar = Fish()
oscar.get_size()  #=> I'm a big fish
```

```
# change the size
bert = Fish()
bert.__size = "small"
bert.get_size() #=> I'm a big fish

# using setter method
fin = Fish()
fin.set_size("tiny")
fin.get_size() #=> I'm a tiny fish

########################################

Abstraction is the concept of hiding all the implementation of your class away from anything
outside of the class.

class Dog:

    def __init__(self, name):
        self.name = name
        print(self.name + " was adopted.")

    def bark(self):
        print("woof!")


# we don't care how it works just bark
spot = Dog("spot") #=> spot was adopted.
spot.bark() #=> woof!

########################################

Polymorphism is a way of interfacing with objects and receiving different forms or results.
```

```
class Animal:

    def __init__(self, name):
        self.name = name
        print(self.name + " was adopted.")

    def run(self):
        print("running!")


class Turtle(Animal):

    def __init__(self):
        super().init

    def run(self):
        print("running slowly!")


# we get back an interesting response
tim = Turtle("tim") #=> tim was adopted.
tim.run() #=> running slowly!

########################################
```

---

1. Inheritance
Object-oriented languages that support classes almost always support the notion of "inheritance." Classes can be organized into hierarchies, where a class might have one or more

parent or child classes. If a class has a parent class, we say it is derived or inherited from the parent class and it represents an “IS-A” type relationship. That is to say, the child class “IS-A” type of the parent class.

Therefore, if a class inherits from another class, it automatically obtains a lot of the same functionality and properties from that class and can be extended to contain separate code and data. A nice feature of inheritance is that it often leads to good code reuse since a parent class’ functions don’t need to be re-defined in any of its child classes.

Consider two classes: one being the superclass—or parent—and the other being the subclass—or child. The child class will inherit the properties of the parent class, possibly modifying or extending its behavior. Programmers applying the technique of inheritance arrange these classes into what is called an “IS-A” type of relationship.

Example: For instance, in the animal world, an insect could be represented by an Insect superclass. All insects share similar properties, such as having six legs and an exoskeleton. Subclasses might be defined for grasshoppers and ants. Because they inherit or are derived from the Insect class, they automatically share all insect properties.

---

2. Encapsulation

The word, “encapsulate,” means to enclose something. Just like a pill "encapsulates" or contains the medication inside of its coating, the principle of encapsulation works in a similar way in OOP: by forming a protective barrier around the information contained within a class from the rest of the code.

In OOP, we encapsulate by binding the data and functions which operate on that data into a single unit, the class. By doing so, we can hide private details of a class from the outside world and only expose functionality that is important for interfacing with it. When a class does not allow calling code access to its private data directly, we say that it is well encapsulated.

Example: Elaborating on the person class example from earlier, we might have private data in the class, such as "socialSecurityNumber," that should not be exposed to other objects in the program. By encapsulating this data member as a private variable in the class, outside code would not have direct access to it, and it would remain safe within that person’s object.

If a method is written in the person class to perform, say, a bank transaction called "bankTransaction()," that function could then access the "socialSecurityNumber" variable as necessary. The person’s private data would be well encapsulated in such a class.

---

3. Abstraction
Often, it’s easier to reason and design a program when you can separate the interface of a class from its implementation, and focus on the interface. This is akin to treating a system as a “black box,” where it’s not important to understand the gory inner workings in order to reap the benefits of using it.

This process is called “abstraction” in OOP, because we are abstracting away the gory implementation details of a class and only presenting a clean and easy-to-use interface via the class’ member functions. Carefully used, abstraction helps isolate the impact of changes made to the code, so that if something goes wrong, the change will only affect the implementation details of a class and not the outside code.

Example: Think of a stereo system as an object with a complex logic board on the inside. It has buttons on the outside to allow for interaction with the object. When you press any of the buttons, you're not thinking about what happens on the inside because you can't see it. Even though you can't see the logic board completing these functions as a result of pressing a button, it's still performing them., albeit hidden to you.

This is the concept of abstraction, which is incredibly useful in all areas of engineering and also applied to great effect in object-oriented programming.

---

4. Polymorphism
In OOP, polymorphism allows for the uniform treatment of classes in a hierarchy. Therefore, calling code only needs to be written to handle objects from the root of the hierarchy, and any object instantiated by any child class in the hierarchy will be handled in the same way.

Because derived objects share the same interface as their parents, the calling code can call any function in that class’ interface. At run-time, the appropriate function will be called depending on the type of object passed leading to possibly different behaviors.

Example: Suppose we have a class called, “Animal” and two child classes, “Cat,” and “Dog.” If the Animal class has a method to make a noise, called, “makeNoise,” then, we can override the "makeNoise" function that is inherited by the sub-classes, "Cat" and "Dog," to be “meow” and “bark,” respectively. Another function can, then, be written that accepts any Animal object as a parameter and invokes its "makeNoise" member function. The noise will be different: either a “meow” or a “bark” depending on the type of animal object that was actually passed to the function.

---

ok) 1. abstraction
encapsulation
inheritance
polymorphism

ok) 2. which of the four fundamental features of the object-oriented programming could be thought of as removing unnecessary
details, allowing the user to focus only on what is necessary? (put NA if it applies to none to them)
(abstraction)

3. Which of the four fundamental features of object-oriented programming essentially means we can create classes from old classes, and the now ones inherent aspects of the old one?

ok) 4. We define a subclass by using the same class keyword but with the child class name insidee parentheses.
(False)

5. Which of these are ways you could refer to original and new class pairings?
superclass/subclass
parent/child

base class/drived class

ok) 6. I have created a class, beverage, that has a method called drink. I then create a class from that class, called coffee.
Ihe coffee also has the drink method, even though I didn't explicitly create it in the class. This is because of one of the four fundamental features called?
(inheritance)

ok) 7. Generally speaking, attributes are directly available in Python.
(True)

ok) 8. Integer objects, such as 7 and 8, are of the same ___, which is why they have the same methods.
(class)

9. A class is a ___ for an object.
a. accessor (correct)
b. blueprint(wrong)
c. mold(wrong)
d. set of instructions(wrong)

ok) 10. Integer objects contain attributes and methods. One such attribute would be the multiply method.
Correct (False)

ok) 11. Which is the best description of how we would add a methods to a subclass?
(When we create a new class from an old one, we can a method just like we would for a parent)

ok) 12. Class and object are essentially the same concept
Correct (False)

13. We use the __init__() method if we want to
    a. assign object methods at creation time (try)
    b. define methods for subclasses(wrong)
    c. assign object attributes at creation time
    d. create an instance of that type

ok) 14. We access both object and class attributes using dot notation.
    (true)

15. car()
    def_init_

ok) 16. Parent objects can inherit from multiple child classes.
    (False)

ok) 17. based on the above code, what will be printed?
    (Mooo)

18. Which of the following is NOT true of using super()?
try) Using super() undermines the principle of inheritance
    (maybe) If the definition of the parent changes, using super will ensure that the attributes and methods will reflect that change.
    a. Using super() undermines the principle of inheritance
    b. It can reduce the amount of code you need to write
    c. All of these are turn (wrong)
    d. If the definition of parent changes, using super will ensure that the attributes and methods will reflect that change

---

ok)1. could be thought of as a mold that creates objects
class

ok)2. Objects are a type of data structure that contain both data and code
data;code

3. Objects have
type, a value, a reference count
variable, function, list, tuple, dictionary, set

ok)4. go and get values of attributes Getters
   change the state of the object Setters

5. Which of the four fundamental features of object-oriented programming essentially means what an object does when there is a method call depends on the class of the object? (put NA if it applies to none of them)
wrong: inheritance
maybe encapsulation

ok)6. have a list of objects of many classes that represent shapes. I run a loop on that list, and use a .draw() method that appropriately draws shapes from these different classes. This is an example of which of the four object-oriented pillars?
Polymorphism

ok)7. Which of the four fundamental features of object-oriented programming essentially means objects contain data and functions? (put NA if it applies to none of them)
Encapsulation

ok)8. We can override any methods, including __init__()
True

ok)9. Yum

ok)10. You find a class that does almost what you need. Inheritance would come to play if you did which of the following?

Create a new class from an existing

11.
```
class Cow:
    def speak(self):
        print("Mooo")
    def eat(self):
        print("Yum")

class Holstein(Cow):
    def speak(self):
        print("MMMOOOOO!")

Bessie = Holstein()
Bessie.speak()
```
MMMOOOOO!

ok)12. We use super() to call a _____ method
parent

ok)13. A Heffer is a Holstein is a Cow

ok)14.
```
class Cow:
    def speak(self):
        print("Moo")
    def eat(self):
        print("Yum")

class Holstein(Cow):
    def talk(self):
        super().speak()
```

```
issubclass(Cow, Holstein)
```

```
False
```

ok)15. Classes cannot contain multiple attributes without instantiation of multiple objects
False

ok)16.

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

```
Cat('Fred','Burmese')
```

ok)17. I have an object, baseball. baseball.throw() would be an example of an attribute.
False

---

ok)1. Based on the above code, what will be printed, and why?
   I have a cow named Bessie; the code is fine

ok)2. A Salesman is an Employee is a Person

3. Based on the above code, which of the following is correct?
wrong)Chef is a Professor
maybe none

ok)4. go and get values of attributes Getter

change the state of the object Setter

ok)5. Integer objects contain attributes and methods. One such attribute would be the multiply method.
False

ok)6. could be thought of as a mold that creates objects
class

ok)7. At the most basic level, objects are a data structure that contain both _____ and _____.
data; code

ok)8. Which of the four fundamental features of object-oriented programming essentially means we can create classes from old classes, and the new ones inherent aspects of the old one? (put NA if it applies to none of them)
inheritance

ok)9. I have a list of objects of many classes that represent shapes. I run a loop on that list, and use a .draw() method that appropriately draws shapes from these different classes. This is an example of which of the four object-oriented pillars?
Polymorphism

ok)10. Which of the four fundamental features of object-oriented programming essentially means objects contain data and functions? (put NA if it applies to none of them)
Encapsulation

11. A function within a class definition is an
object function
maybe method
methods are functions built in to the class definiton of an object

ok)12. .describe() is an attribute of a DataFrame object
False

ok)13. Given this code, what is the type of mycar?
__main__.car

ok)14. Functions in a class or object are called
method

15. Which of the following is NOT true in regards to methods?
wrong: When you create a method you should put 'self' in the parentheses of the method name
wrong: Methods can be created using def just like regular functions
maybe: Child classes cannot have methods of the same name as the parent class

16. The exclaim method prints out "I'm a car!" even though we didn't create it in our Prius definition. Why would it be able to do this?
The method exclaim is a base Python method which it my_car has by default

ok)17. Based on the above code, which of the following is produced?
MMMMOOOOO!

ok)18. We define a subclass by using the same class keyword but with the child class name inside parentheses
False

---

ok)1. I have created a class, beverage, that has a method called drink. I then create a class from that class, called coffee. The coffee class also has the drink method, even though I didn't explicitly create it in the class. This is because of one of the four fundamental features called

(inheritance)

ok)2. The exclaim method prints out "I'm a car!" even though we didn't create it in our Prius definition. Why would it be able to do this?
(It inherits the method from the car class)

ok)3. Yum

ok)4. We access both object and class attributes using dot notation
(True)

ok)5. Given the above code, which is the correct way to instantiate a cat?
(Cat('Fred', 'Burmese'))

cc)6. .describe() is an attribute of a DataFrame object
(False)

7. We use the __init__() method if we want to
wrong(assign object methods at creation time)
wrong(assign object attributes at creation time)
maybe(create an instance of that type)

ok)8. Generally speaking, attributes are directly available in Python
(True)

9. Method resolution order determines
The Python Method Resolution Order defines the class search path used by Python to search for the right method to use in classes having multi-inheritance.
wrong(class)
maybe(inheritance)

ok)10. Which of the following is true?

A Heffer is a Holstein is a Cow

ok)11. Parent objects can inherit from multiple child classes
(False)

ok)12. Integer objects, such as 7 and 8, are of the same ______, which is why they have the same methods.
(class)

13. A class is a(n) _________ for an object
Which of the following is not appropriate to fill in the blank?
wrong(mold)
wrong(blueprint)
try(set of instructions)

14. Objects have
A type, A value, A reference count, unique id

ok)15. Which is the best description of how we would add a method to a subclass?
(When we create a new class from an old one, we can add a method just like we would for a parent)

ok)16. Which of the four fundamental features of object-oriented programming could be thought of as removing unnecessary details, allowing the user to focus only on what is necessary? (put NA if it applies to none of them)
(Abstraction)

ok)17. Question 17 options:
In alphabetical order, what are the four fundamental features of object-oriented programming?

ok)18. Which of the four fundamental features of object-oriented programming essentially means what an object does when there is a method call depends on the class of the object? (put NA if it applies to none of them)
(polymorphism)

Which of the following is not appropriate to fill in the blank?

Which portion(s) of this code are incorrect or incomplete?
car()
def_init_
(self, name):

Packages DTSC 575

pip install flask == 0.9 >= (min version)
pip install –upgrade <package name>

Virtual environment

Py Sci and Stats

```
import statistics
x = [1, 2, 3]
statistics.mean(x)
```

```
import numpy as np
y = np.array([1, 2, 3])
y.mean()

import seaborn as sns
tips = sns.load_dataset('tips')
tips.head()

np.corrcoef(tips.total_bill, tips.tip)

import scipy
r, p = scipy.stats.pearsonr(x, y)
r

import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels.api as sm
import statsmodels.formula as smf
from patsy import dmatrices

df = sm.datasets.get_rdataset("Guerry", "HistData").data

df.head()
```

Are literacy rates associated with lottery wagers?
We're interested in the following:

- Department
- Lottery = per capita wager on Royal Lottery
- Literacy = Percent of military conscripts who can read & write
- Wealth = Per capita tax on personal property

```
df2 = df[['Department', 'Lottery', 'Literacy', 'Wealth', 'Region']]
df2.head()

df2.describe()
```

Design matrices
Generally statsmodels requires two design matrices: endog and exog

* Endog = endogenous
    * Dependent/response variables
* Exog = exogenous
    * Independent/predictor variables

---

1. Dependent/response variables Exogenous
   Independent/predictor variables Endogenous

2. Statsmodels generally requires ___ matrices
   (2)

3. When you run an ANOVA the output does NOT tell you which of the following?
   F-statistic
   Degrees of freedom
   Overall significance of the model
   *Multiple comparisons of means

4. ANOVA stands for
   (analysis of variance)

5. ANOVA is an omnibus test
   (True)

6. By default, scikit-learn and statsmodels create equivalent logistic regression models
   (False)

7. Scatterplots are useful to visualize relationships you might investigate in linear regression
   (True)

8. Which of the following is how we would create a logistic regression model?
   (sm.Logit())

9. Boxplots are useful in comparing mean differences across groups
   (True)

10. Scatterplots are useful for comparing mean differences across groups
     (False)

11. You can run some statistics using NumPy and Pandas methods.
    (True)

12. For relatively low-level stats, it generally doesn't matter if you run them in statsmodels, the statistics library, or the other ways we saw
    (True)

W)13. Flask install pip == .09 Would install Flask version .09
    (It's pip version)

14. Anaconda comes with some packages, like NumPy and Pandas, preinstalled
    (True)

15. pip comes with the standard Python installation
    (True)

w)16. Which of the following are common ways of installing packages?
    conda
    pip
    install.packages()
    takeout(install.py)

---

1. To run any stats you must first import statistics
   (False)

3. Which of these might you want to add to the default statmodels regression model?
   (Constant)

4. import seaborn as sns

5. The default linear regression model will be equivalent if you run it in R and statsmodels
   (False)

6. Multiple comparisons are a post-hoc test to indicate where a significant difference between groups might be found
   (True)

7. The following will compare differences in traffic across days
   multicomp.pairwise_tukeyhsd(df.day, df.traffic)
   (False)

8. OLS can be used for regression and ANOVA
   (True)

9. Independent samples t-tests compare the mean across three or more groups

(False)

10. We have a dataframe, df, and a variable, gender. We have the following code. Which is true?
    from statsmodels.stats.weights import ttest_ind
    ttest_ind(df.gender)
    (ttest_ind() was not created properly; there will be an error)

11. Which of the following can you do with pip?
    uninstall packages
    upgrade packages
    install packages

12. Packages outside of base Python never come with any Python installation.
    (False)

14. Which of the following are common ways of installing packages?
    conda
    pip

15. Which ways do we frequently import statsmodels?
    (import statsmodels.api as sm)
    (import statsmodels.formula as smf)

16. You cannot have two endogenous variables
    (False)
