OBJECTS

NAME SPACES

DUNDERS

- Double Underscores
  - `__name__`
  - `__init__`
  - `__variable`

PYTHON NAMESPACES

- Default namespaces are:
  - `__builtins__`
    - Functions, errors, types that are built-in to Python
  - `__main__`
    - Initially empty; where programmer defined functions exist
- `__name__` to see current namespace
- `dir(__aNamespace__)` to see functions of that namespace
  - i.e. `dir(__builtins__)`

DOC STRINGS

- __doc__ to view information about modules
  - Not always defined
    - i.e. `main.__doc__` will return `NameError`
  - Can be defined

```
import math

math.__doc__
```

Will return:

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

EDITING DOC STRINGS

- If `main.__doc__` is empty, can it be edited?

EDITING DOC STRINGS

```
def main():
    """an edited docstring"""
    #nothing in main except
    docstring
```

```
def myFunction():
    '''prints "hello" to
    user'''
    print("hello")
```

What will return now?

```
main.__doc__
```

```
myFunction()
myFunction.__doc__
```

IMPORTING MODULES

```
import math
```

vs.

```
from math import *
```

Which namespace will the `math` functions be found?

- __builtins__ vs __main__ namespaces
- When function is called, Python will search inside-out in namespaces
  - Local, main, then builtins namespace

CLASSES AND CONSTRUCTORS

OBJECTS

- Objects – individual data items that are manipulated to solve a problem
- Classes – what an object will "look" like; what an object knows about itself and what it can do

CLASSES

- What an object will "look" like
  - Template
  - Each object belonging to a specific class will have same type of data, structure
- What an object knows about itself and what it can do
  - Instance data
  - Methods

```
class Classname:
    '''Classname docstring'''
    #methods and stuff

    ...
```

CONSTRUCTORS

- Method required by all classes to create the data object

```
class Classname:
  '''Classname docstring'''

  def __init__(self, data,…):
    self.data = data
  ...
```

INSTANCE DATA

If run in a cell, `self.data` will return data that was passed in the `__init__` method of our class

- Given that information, can we edit `self.data` directly?
  - i.e. `self.data = 'some other value'`

Should you be able to edit `self.data` directly?

ENCAPSULATION

- To keep instance data from being edited 'out of scope', it can be *encapsulated* using dunder
  - Make instance variables private

```
def __init__(self, data,…):
    self.__data = data
```

ENCAPSULATION

Before using dunder:

```
myObject = Classname(someData)

myObject.data
```

Will Return:

```
'data'
```

After using dunder:

```
myObject = Classname(someData)

myObject.data
```

Will Return:

```
'Classname' object has no attribute 'data'
```

CLASS METHOD

- Printing a string ‘hello’

How can you view Classname’s `sayHello` docstring?

```
class Classname:
    ...
    def sayHello(self):
        '''this function prints hello'''
        print('hello')
    ...
```

TYPE HINTS

How do you know what type of data to pass in if no type is specified?

TYPE HINTS

- Type hints help with readability when writing and using functions
- Parameter type – after variable name, use a colon and data type
- Return type – after function data close parenthesis, use arrow (->) and data type

TYPE HINT EXAMPLE

```
class Classname:
  ...
  def doubleInt(self, anInt:int)->int:
    doubled = 2*anInt
    return doubled
  ...
```

TYPE ERRORS

- Type hints aren't required or enforced by Python
  - Many third-party tools and IDEs use them though
- So, by themselves, type hints don't enforce data type in programs
  - This can cause confusion and irritation within the programmer

TYPE ERRORS

- The most basic way to check if parameter is of correct type is to see if it `isinstance()`

TYPE ERRORS

```
class Classname:
    ...
    def doubleInt(self, anInt:int)->int:
        if not isinstance(anInt, int):
            raise TypeError("Value must be int type.")
        doubled = 2*anInt
        return doubled
    ...
```

PYTHON SPECIAL METHODS

SPECIAL METHODS

- AKA ‘Magic Methods’
  - AKA ‘Dunder Methods’ since they also start and end with double underscores

SPECIAL METHODS

- __getitem__ = indexing
- __add__ = addition
- __str__ = object as string
- __eq__ = equal
- __ne__ = not equal
- __lt__, __le__ = less than, less than or equal
- __gt__, __ge__ = greater than, greater than or equal

USING SPECIAL METHODS

How would this change for an object of Classname?

What will the codes print?

```
anInt = 5

anInt.__add__(3)
print(anInt.__str__)
```

Returns:

```
<method-wrapper '__str__' of
int object at 0xlocation>
```

```
anInt = 5

anInt.__add__(3)
print(anInt.__str__())
```

Returns:

```
5
```

USING SPECIAL METHODS

- Special methods can be defined with more useful information for our classes

```
class Classname:
    ...
    def __str__(self):
        return self.__specificInformation
```