ΠΕΡΙΕΧΟΜΕΝΑ:

1. Ορισμός Enumeration
2. Επιπλέον λειτουργικότητα
3. Περισσότερα για την κατασκευή
4. Ειδικά enumerations

ΠΡΟΑΠΑΙΤΟΥΜΕΝΑ:

1. Python Advanced: Μάθημα 4, 5 - Decorators

Ανδρέας Γ.

Σμαραγδένιος Χορηγός Μαθήματος

Δημήτρης Δ.

Σμαραγδένιος Χορηγός Μαθήματος

ΜΑΘΗΜΑ 2.4: Το module enum

1. Ορισμός Enumeration

- Ένας **απαριθμητής (enumeration)** είναι ένας τύπος δεδομένων που τα αντικείμενα του, παίρνουν ένα μικρό πλήθος τιμών.
- Δηλώνουμε έναν απαριθμητή, κληρονομώντας την κλάση Enum του module enum:

```
from enum import Enum
class enum_name(Enum):
   CONSTANT1 = val1
   CONSTANT2 = val2
   CONSTANT3 = val3
```

  - **Κάθε μέλος** του απαριθμητή, δηλώνεται με τον παραπάνω (έμμεσο) τρόπο ως **στατική σταθερά** και έχει μία τιμή της αρεσκείας μας (διαφορετική από τα άλλα μέλη). Ο τύπος δεδομένων κάθε σταθεράς είναι το όνομα του enum/tion.
- Χρησιμοποιούμε μεταβλητές που παίρνουν τιμές από τον απαριθμητή ως εξής:
  - Εντολή ανάθεσης που δίνουμε τιμή σε μία μεταβλητή:

```
variable = enum_name.CONSTANTx;
```

  - αλλά και να την τυπώσουμε. Προσοχή! Αν τυπώσουμε τη μεταβλητή, τυπώνεται το όνομα της σταθεράς.

```
print(variable); # prints enum_name.CONSTANTx
```

Παρατήρηση:
- Η τιμή μιας σταθεράς έχει καθαρά εσωτερική λειτουργία.
- Μας ενδιαφέρουν μόνο τα ονόματα των σταθερών.

Παράδειγμα 1: simple_enum

```
from enum import Enum

class Days(Enum):
   MONDAY = 1
   TUESDAY = 2
   WEDNESDAY = 3
   THURSDAY = 4
   FRIDAY = 5
   SATURDAY = 6
   SUNDAY = 7

today = Days.MONDAY
print(today) # prints Days.MONDAY
print(today.name) # prints MONDAY
print(today.value) # prints 1
print(type(today)) # prints <enum 'Days'>
print(isinstance(today, Days)) # True
```

Γιατι είναι χρήσιμα τα enumerations:
- Τα enumerations χρησιμοποιούνται, ώστε ο κώδικας να γίνει πιο αναγνώσιμος. Επίσης μέσω των σταθερών, πεπερασμένων τιμών ελαχιστοποιούνται λάθη κατά τον προγραμματισμό.
- Επιδέχονται και περαιτέρω παραμετροποίησης, όπως θα δούμε στη συνέχεια.

Υποστηρίζονται τα εξής:

- **Επανάληψη** (είναι iterable)

```
# iterable
for day in Days:
   print(day)
```

- Είναι **hashable** (π.χ. μπορεί να χρησιμοποιηθεί ως κλειδί λεξικού)

```
# hashable
workload = {Days.MONDAY: "a lot",
            Days.TUESDAY: "a little",
            Days.SUNDAY: "nothing"}
print(workload)
```

- Εκτός του συνηθισμένου τρόπου για να πάρουμε μια τιμή του enumeration (π.χ. Days.MONDAY), καταφέρνουμε το ίδιο, μέσω σχετικής θέσης, είτε πρόσβασης ως λεξικό:

```
# get an enum member
print(Days.WEDNESDAY) # usual way
print(Days["WEDNESDAY"]) # by member name
print(Days(3)) # by member value
```

Παράδειγμα 2: functionality.py

- Οι **τιμές είναι συγκρίσιμες** μέσω == ή !=, αλλά δεν είναι συγκρίσιμες με άλλους σχεσιακούς τελεστές (<, >, <=, >=)

```
# comparisons
print(Days.MONDAY==Days.MONDAY)
print(Days.MONDAY!=Days.TUESDAY)
try:
   print(Days.MONDAY<Days.TUESDAY)
except TypeError:
   print("can't compare them")
```

  - Επίσης δεν θα λειτουργήσει η σύγκριση μέλους και τιμής (π.χ. Days.MONDAY==1 δεν είναι True)
- **Δεν επιτρέπεται να κληρονομηθούν** από άλλες κλάσεις, αλλά επιτρέπεται να έχουν μεθόδους

Παράδειγμα 3: comparable.py

```
@functools.total_ordering
class Days(Enum):
   MONDAY = 1
   TUESDAY = 2
   WEDNESDAY = 3
   THURSDAY = 4
   FRIDAY = 5
   SATURDAY = 6
   SUNDAY = 7
   def __gt__(self,other):
      return self.value > other.value
```

```
print(Days.MONDAY > Days.WEDNESDAY)
print(Days.MONDAY >= Days.WEDNESDAY)
print(Days.MONDAY < Days.WEDNESDAY)
print(Days.MONDAY <= Days.WEDNESDAY)
```

Επιπλέον συντακτικό για τη δήλωση ενός Enum:

- Μπορούμε να ορίσουμε να γίνεται αυτόματη απόδοση τιμών σε κάθε μέλος του απαριθμητή με την auto()
- **Η auto() αποδίδει αύξουσα αρίθμηση**, ξεκινώντας από το 1, σε κάθε μέλος του απαριθμητή.

Παράδειγμα 4: auto_values.py

```
class Days(Enum):
    MONDAY = auto()
    TUESDAY = auto()
    WEDNESDAY = auto()
    THURSDAY = auto()
    FRIDAY = auto()
    SATURDAY = auto()
    SUNDAY = auto()
```

```
for day in Days:
    print(day.name, day.value)
```

- Επιτρέπεται να έχουμε ίδιες τιμές σε μέλη (αν και δεν συνηθίζεται)
- Ενώ εξασφαλίζουμε ότι οι **τιμές είναι διαφορετικές**, με τον **decorator κλάσης @unique**

Παράδειγμα 5: unique_values.py

```
@unique
class Days(Enum):
    MONDAY = 1
    TUESDAY = 2
    ...
```

- **Η κλάση Enum είναι Callable.**
  - 1ο όρισμα: όνομα του enumeration
  - 2ο όρισμα: συμβολοσειρά. Χωρισμένα με κενό τα ονόματα των μελών του enumeration

Παράδειγμα 6: callable.py

```
Days = Enum("Days", "MONDAY TUESDAY WEDNESDAY
                    THURSDAY FRIDAY SATURDAY SUNDAY")
for day in Days:
    print(day.name, day.value)
```

Παρατήρηση:

- Είδαμε στην προηγούμενη διαφάνεια, ότι μπορούμε να έχουμε ειδικές μεθόδους (dunder), αλλά το Enum είναι απλά μία κλάση και έτσι μπορούμε να έχουμε μεθόδους (που θα αναφέρονται στα μέλη) ή ακόμη και στατικές μεθόδους (μεθόδους κλάσης, βλ. Python Advanced - Μάθημα 5)

Παράδειγμα 7: methods.py

```
class Days(Enum):
    MONDAY = 1
    TUESDAY = 2
    WEDNESDAY = 3
    THURSDAY = 4
    FRIDAY = 5
    SATURDAY = 6
    SUNDAY = 7
    def mood(self):
        if self.value <= 5:
            return "bad"
        else:
            return "good"
    @classmethod
    def best_day(cls):
        return cls.SUNDAY
```

```
for day in Days:
    print(day, day.mood())

print(Days.best_day())
```

IntEnum είναι επιπλέον, υποκλάση της int.

- Τα μέλη της μπορούν να χρησιμοποιηθούν οπουδήποτε μπορούν να χρησιμοποιηθούν ακέραιοι

Παράδειγμα 8: int_enum.py

```
class Days(IntEnum):
    MONDAY = 1
    …

week_jobs = ["some", "more work", "much more work"]

print(week_jobs[Days.MONDAY])
print(week_jobs[Days.TUESDAY])
```

IntFlag είναι επιπλέον, υποκλάση της int.

- Αλλά δουλεύουν οι τελεστές πράξεων bit μεταξύ των ακεραίων τιμών: &, |, ^, ~

Παράδειγμα 8: int_flag_enum.py

```
class Perm(IntFlag):
    READ = 1
    WRITE = 2
    EXECUTE = 4
```

```
RW = Perm.READ | Perm.WRITE
print(Perm.READ in RW, RW)

RWX = RW | Perm.EXECUTE
print(RWX)

RX = RWX & ~ Perm.WRITE
print(RX)
```

Flag:

- Οι τελεστές πράξεων bit μεταξύ των ακεραίων τιμών: &, |, ^, ~ έχουν επίσης οριστεί (όπως η IntFlag)

Συστάσεις για τη Flag:

- Να χρησιμοποιούμε την auto() για να αποδίδονται τιμές που είναι δυνάμεις του 2.
- Μπορούμε να δίνουμε συνδυασμούς τιμών (που αυτόματα δεν θα είναι δυνάμεις του 2), χρησιμοποιώντας την OR ('|')
- (Αναφέρεται ότι οι IntEnum και IntFlag καλό θα είναι να μην χρησιμοποιούνται και να πρωτιμώνται μόνο οι Enum και Flag, διότι είναι κάπως ασυνεπές να είναι υποκλάση του ακεραίου ένα enumeration)

Παράδειγμα 9: flag_enum.py

```
class Color(Flag):
    RED = auto()
    BLUE = auto()
    GREEN = auto()
    WHITE = RED|GREEN|BLUE

for item in Color:
    print(item, item.value)
```