ΠΕΡΙΕΧΟΜΕΝΑ:

1. Τεκμηρίωση Κώδικα
2. Έλεγχος
    1. Έλεγχος Συνάρτησης
    2. Έλεγχος Κλάσης
3. Data Structures: Γράφος (Graph)
4. Data Project: Συσχετίσεις Part 2

Σπύρος Παπαγιάκουμος

Σμαραγδένιος Χορηγός Μαθήματος

Νίκος Θ.

Ασημένιος Χορηγός Μαθήματος

- **Σχολιασμός Κώδικα:** Απευθύνεται σε προγραμματιστή και πρέπει να είναι σύντομος και περιεκτικός. Χρησιμοποιούμε τη #. Περιλαμβάνει π.χ.:
  - Σημειώσεις (π.χ. ToDos)
  - Σχεδιασμός Κώδικα (π.χ. καθορισμός βασικών βημάτων σχεδιάζοντας την αρχική εφαρμογή)
  - Βήματα περιγραφής αλγορίθμων
  - Στην έκδοση 3.5 προστέθηκε (ως σχόλιο) η επιστρεφόμενη τιμή (Type Hinting) (βλ. παράδειγμα 1)
- **Τεκμηρίωση Κώδικα:** Απευθύνεται σε προγραμματιστή, αλλά και σε χρήστη του κώδικά μας ή πελάτη μας. Χρησιμοποιούνται τα docstrings (σχόλια πολλών γραμμών τεκμηρίωσης) που τοποθετούνται σε συγκεκριμένα σημεία του προγράμματος μας. Από αυτά μπορούν, αυτοματοποιημένα, να παραχθεί τεκμηρίωση, χρησιμοποιώντας πακέτα. Δύο συνηθισμένες συμβάσεις:
  - Αμέσως μετά το όνομα της κλάσης, περιγράφονται τα μέλη και οι μέθοδοι της κλάσης
  - Αμέσως μετά την πρώτη γραμμή του ορισμού μιας συνάρτησης ή μίας μεθόδου, ακολουθεί η περιγραφή των ορισμάτων, η συμπεριφορά και η επιστρεφόμενη τιμή (βλέπε παράδειγμα 2)

  Λεπτομέρειες του πρότυπου περιγραφής συχνά αλλάζουν ανάλογα με το πακέτο που δουλεύουμε για την αυτόματη δημιουργία documentation από τα docstrings.

Παράδειγμα 1: comment.ret.value.py

```
def full_name(first_name, last_name) -> str:
    return f"{first_name} {last_name}"

print(full_name("John", "Wick"))
```

Παράδειγμα 2:

Στο ακόλουθο παράδειγμα (από realpython.com) έχει πραγματοποιηθεί αναλυτική τεκμηρίωση μιας κλάσης:

```
class Animal:
    """
    A class used to represent an Animal


    ...

    Attributes
    ----------
    says_str : str
        a formatted string to print out what the animal says
    name : str
        the name of the animal
    sound : str
        the sound that the animal makes
(cont'd)
```

- Ο **έλεγχος (testing)** θεωρείται πολύ σημαντικό κομμάτι της ανάπτυξης λογισμικού:
  - Σε μεγάλες εταιρίες, υπάρχει ειδικό τμήμα για testing
  - Μας κατοχυρώνει (όσο γίνεται, γιατί δεν μπορεί να γίνει τελείως ολοκληρωμένα) ότι ο κώδικας δεν θα έχει απροσδόκητα λάθη.
- Ήδη έχουμε κάνει έλεγχο στα προγράμματά μας:
  - Το τρέχουμε και διαπιστώνουμε ότι όλα πηγαίνουν καλά.
  - Αυτός ο **έλεγχος (χειροκίνητος, manual)** είναι πάντα ημιτελής και υπάρχει ισχυρή περίπτωση να οδηγήσει σε αστοχίες.
- Υπάρχουν δύο βασικά είδη ελέγχου:
  - **Unit testing (έλεγχος μονάδων):** Εδώ ελέγχουμε αυτόνομα τμήματα κώδικα όπως μια συνάρτηση ή μία κλάση
  - **Integration testing (έλεγχος ολοκλήρωσης):** Ελέγχεται αν τα αυτόνομα κομμάτια αλληλεπιδρούν σωστά στην ολοκληρωμένη εφαρμογή.
- Υπάρχουν πολλά πακέτα για τον έλεγχο:
  - Θα μελετήσουμε το **unittest** το οποίο είναι built-in στην Python.
  - Υπάρχουν πολλά ακόμη πακέτα (PyTest, nose, …)

Αν και θα μελετήσουμε το unittest, θα κάνουμε μια αναφορά στην εντολή assert:

- Ο έλεγχος μπορεί να γίνει μέσω assertions (διαβεβαιώσεων).
- Η built-in συνάρτηση assert που παίρνει δύο ορίσματα:
  - Το πρώτο όρισμα είναι ένας έλεγχος (Boolean αποτέλεσμα)
  - Το δεύτερο όρισμα είναι ένα μήνυμα λάθους.
  - Αν ο έλεγχος αποτύχει, τότε το πρόγραμμα τερματίζει με την εξαίρεση AssertionError

Παράδειγμα 3: assert.py

```
def my_sum(sth):
    s = 0
    for item in sth:
        s += item
    return s
assert my_sum((1,2,3)) == 6
```

Παράδειγμα 4: assert2.py

```
def my_avg(sth):
    assert len(sth)!=0, "iterable is empty!"
    s = 0
    for item in sth:
        s += item
    return s/len(sth)
print(my_avg(()))
```

- Για να ελέγξουμε μια συνάρτηση:
  - Αποφασίζουμε τα **unit tests** (μεμονωμένες περιπτώσεις εισόδου και επιθυμητής εξόδου)
  - Όλα τα unit tests μαζί συγκροτούν μια περίπτωση ελέγχου **(test case)**

Π.χ.: Θα ελέγξουμε τη my_sum (Μάθημα 13, args.py) γνωρίζοντας ότι θέλουμε να δουλεύει σωστά όταν καλείται με 0 ή 3 ακεραίους

Οι έλεγχοι θα γίνουν ως εξής (test_args.py):

```
import unittest
from args import my_sum # import function to be tested

class MySumTestCase(unittest.TestCase):
    def test_1(self):
        self.assertEqual(my_sum(1, 2, 3), 6)

    def test_2(self):
        self.assertEqual(my_sum(), 0)
```

Παρατηρήσεις:

- Ακολουθείται ένα αυστηρό συντακτικό, ορίζοντας απλά μία κλάση και ορίζοντας τους ελέγχους.
- Έπειτα τρέχοντας το αρχείο, εκτελούνται οι έλεγχοι με τα διαγνωστικά τους μηνύματα (βλ. βίντεο)

Κανόνες:

- import το πακέτο unittest
- import τη συνάρτηση από τον κώδικά μας
- Κατασκευάζουμε μια κλάση:
  - Το όνομα της κλάσης, είναι της αρεσκείας μας.
  - Υποχρεωτικά κληρονομούμε το unittest.TestCase
  - Ορίζουμε τα unit tests ως μεμονωμένες μεθόδους. Πρέπει να ξεκινάνε υποχρεωτικά με test και underscore (το υπόλοιπο λεκτικό (στο π.χ. είναι 1 και 2) είναι της αρεσκείας μας.
  - Στο σώμα του test έχουμε κάποια assert από τις ακόλουθες:

Έλεγχος		Έλεγχος	
assertEqual(a, b)	a == b	assertIsNone(x)	x is None
assertNotEqual(a, b)	a != b	assertIsNotNone(x)	x is not None
assertTrue(x)	x == True	assertIn(a, b)	a in b
assertFalse(x)	X == False	assertNotIn(a, b)	a not in b
assertIs(a,b)	a is b	assertIsInstance (a, b)	isInstance(a, b)
assertIsNot(a, b)	a is not b	assertNotIsInstance (a, b)	not isInstance(a, b)

- Για να ελέγξουμε μια κλάση:
  - Κατασκευάζουμε μια κλάση με τον ίδιο τρόπο όπως στον έλεγχο της συνάρτησης
  - Μπορούμε να ορίσουμε αντικείμενα της κλάσης και άλλα αντικείμενα ελέγχου στη μέθοδο **setUp**
  - Τα αντικείμενα αυτά μπορούμε να τα χρησιμοποιήσουμε στα tests μας (test_teacher.py):

```
import unittest
from teacher import Teacher
class TeacherTestCase(unittest.TestCase):
    def setUp(self) -> None:
        self.bush = Teacher("George", "Bush")
        self.clinton = Teacher("Bill", "Clinton", 1001)
    def test_from_dict(self):
        dict_arg = {
            "first_name": "George",
            "surname": "Bush",
            "teacher_id": -1
        }
        t = Teacher()
        t.from_dict(dict_arg)
        self.assertEqual(self.bush.first_name, t.first_name)
        self.assertEqual(self.bush.surname, t.surname)
        self.assertEqual(self.bush.teacher_id, t.teacher_id)
```

```
def test_to_dict(self):
    dict_arg = {
        "first_name": "George",
        "surname": "Bush",
        "teacher_id": -1
    }

    self.assertEqual(self.bush.to_dict()["first_name"],
                     dict_arg["first_name"])
    self.assertEqual(self.bush.to_dict()["surname"],
                     dict_arg["surname"])
    self.assertEqual(self.bush.to_dict()["teacher_id"],
                     dict_arg["teacher_id"])
```

Άσκηση 1:

Ελέγξτε στην κλάση Waiter (Μάθημα 18, άσκηση 7) ότι:

- Η αρχικοποίηση του αντικειμένου δουλεύει σωστά.
- Η συνάρτηση serve δουλεύει σωστά (το πλήθος των πελατών που έχει εξυπηρετήσει μετρίεται σωστά)

Άσκηση 2.1: Κλάση Γράφος

Ο γράφος είναι μία δομή δεδομένων με μη γραμμική διάταξη, ο οποίος αποτελείται από:

- τους κόμβους (κορυφές - vertices)
- τις ακμές (πλευρές - edges)

Ορίστε την κλάση κόμβος (Node):

- Μέλος: Περιγραφή (descr) κόμβου: Συμβολοσειρά
- Μέλος: Γείτονες (neighbors) κόμβου: Λίστα από αναφορές σε κόμβους.

Ορίστε την κλάση γράφος (Graph):

- Μέλος: Λίστα Κόμβων. Να αρχικοποιείται στην κενή λίστα
- add_vertex: Παίρνει ορίσματα την περιγραφή και τη λίστα γειτόνων (προαιρετικό), κατασκευάζει τον κόμβο και τον προσθέτει στη λίστα γειτόνων.
- add_edge: Παίρνει ορίσματα τις περιγραφές δύο κόμβων και προσθέτει την ακμή που τους συνδέει.

Άσκηση 2.2: Ένα παράδειγμα:

Ο γράφος του facebook:

- Κάθε χρήστης είναι και ένας κόμβος
- Υπάρχει ακμή που συνδέει δύο χρήστες ανν αυτοί είναι φίλοι.
- (κατασκευάστε το δίκτυο φίλων του σχήματος)

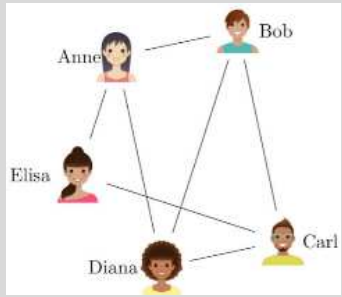

Image From: https://miro.medium.com/max/1044/1*vkyE_OhBWNRqV1umRiISaA.png

Άσκηση 2.3: Διάσχιση πρώτα κατά πλάτος (προχ. => βλ.βίντεο)

Ο αλγόριθμος πρώτα κατά πλάτος (breadth first search) βρίσκει το συντομότερο μονοπάτι σε ένα γράφο (δεδομένου μιας αφετηρίας και ενός προορισμού)

- Αφού τον υλοποιήσετε (ψευδογλώσσα από Wikipedia), βρείτε το συντομότερο μονοπάτι για να γνωρίσει ο Carl την Elisa.

```
1  procedure BFS(G, root) is
2      let Q be a queue
3      label root as discovered
4      Q.enqueue(root)
5      while Q is not empty do
6          v := Q.dequeue()
7          if v is the goal then
8              return v
9          for all edges from v to w in G.adjacentEdges(v) do
10             if w is not labeled as discovered then
11                 label w as discovered
12                 w.parent := v
13                 Q.enqueue(w)
```

Άσκηση 3: Διορθώσεις στις συσχετίσεις

Παρατηρήστε ότι στις ενέργειες που κάναμε στην προηγούμενη έκδοση, δεν ασχοληθήκαμε με:

- Τις διορθώσεις που πρέπει να γίνουν στα μαθήματα, αν διαγραφεί κάποιος καθηγητής ή μαθητής.

Προχωρήστε στις απαραίτητες διορθώσεις, έτσι ώστε όταν διαγράφεται ένας καθηγητής ή μαθητής, να ενημερώνονται αντίστοιχα και τα μαθήματα.

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.
10. Errors should never pass silently.
11. Unless explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.
13. There should be one-- and preferably only one --obvious way to do it.
14. Although that way may not be obvious at first unless you're Dutch.
15. Now is better than never.
16. Although never is often better than *right* now.
17. If the implementation is hard to explain, it's a bad idea.
18. If the implementation is easy to explain, it may be a good idea.
19. Namespaces are one honking great idea -- let's do more of those!
20. ...

Zen of Python