סיכום מונחה עצמים :

תבניות עיצוב :

Adapter - מתאם בין ממשקים שונים , המחלקה החדשה המתאמת תירש את המימוש הישן.

Singleton - כאשר נרצה להשתמש באותה מחלקה במקומות שונים ואנחנו לא רוצים שהיא תתנהג בצורה שונה נבנה סינגלתון .

Sychronized בא לפתור את הבעיה שיכולה להיווצר כשיש הרבה treads אז Sychronized דואג לתעדף את זה ולגרום שלא כולם יכנסו במקביל .

Iterator – כלי למעבר על איברים במבנה נתונים – מצהירים על האיטרטור וע"י פקודת next() נתקדם.

נשתמש כאשר נרצה לעבר על מבנה נתונים ללא אינדקסים.(כל מבנה נתונים ללא key).

Iterabale – אם נממש את המחלקה הזו נאפשר לאובייקט שאנחנו עוברים עליו להיות המטרה ללולאת "for – each" .ובנוסף נבנה לי איטרטור בצורה אוטומטית .(איטרטור לשימוש בשביל לולאת "for – each"). לעומת זאת אם אני ממש את המחלקה Iterator אצטרך לבנות את כל המתודות כגון :()remove, ()HasNext , ()next .

- ההבדל בין iterabel לiterator הוא שiterable הוא ממשק וכאשר תממש אותו הוא מחייב אותך להחזיר iterator .

State - יצירת שינויים על עצם (מחלקה) מחוץ למחלקה – לדוגמא אם ארצה להפעיל מצב ספורט ברכב איהיה חייב לעשות את זה מחוץ למחלקה "רכב" ואז המחלקה הוחלפה.(אפשר להגיד שזה ההתנהגות של האובייקט וניצור אותו חיצונית למחלקה כי זה לא משותף לכולם.)

Observer – קבלת התראה אם חלו שינויים בState של אובייקט שאנחנו עוקבים אחריו.

שלב ראשון - נממש שתי מחלקות מחלקת subject שתייצג את האובייקט שנעקוב אחריו ומחלקת Observer שבעזרת נוכל לעקוב.

שלב שני – במחלקה הכללית (הרחבה יותר) נממש את subject ולכן נצטרך לממש את המתודות של subject ,במימוש המתודות נשלב בין האובייקט לבין הObserver וכן נבצע את המעקב ונדע את השינויים.

Composite – לחלק בעיה לתתי בעיות – ישנה מחלקה אחת גדולה ובתוכה יש תתי מחלקות שפותרות בעיות קטנות ומתנהג כמו עץ.

כמו שרי אלפים שרי מאות ושרי עשרות – הבעיה מתחילה מלמטה ואם לא נפתרת עולה למחלקת האב.

הסבר נוסף : תבנית עיצוב שמטרתה לייצור אובייקט שנבנה על פי שיטה ייצור של כמה תתי אובייקטים שנבנו מכמה תתי אובייקטים עד לבסיס, לדוגמא:

נרצה לבנות את האובייקט חדר, לכן נבנה את האובייקטים של מרכבי החדר ואת האובייקטים שדרכם יוצרים את מרכבי החדר וכך ניצור כל אובייקט על פי האובייקטים שנוצרו מהן

למשל בדוגמא של חדר:

חדר בנוי ממיטה, שולחן , טלויזיה, ארון וכו'..

מיטה בנויה מעץ מזרון וכו..

שולחן בנוי מעץ ברגים וכו..

וכך נבנה את כל האובייקטים מהבסיס עד שנגיע לאובייקט חדר שהוא בנוי מכל האובייקטים האלו.

נשים לב שתהליך בניית האובייקטים יבנה באופן זהה לאירככיה של עץ.

נשתמש בזה כאשר נרצה להרחיב משהו קיים כדי לפתור בעיה מסויימת ולא לכתוב מחלקה שלמה עם מתודות חדשות רק בשביל הבעיה הספציפית הזו.

תכנות על פי חוזה (DBC) – שיטת עיצוב תוכנה המתחשבת ברכיבי חומרה.

בשיטה זו נשתמש בטיפוסים אבסרקטים (טיפוס שהמימוש שלו מוסתר) המקיימים את התנאים ("חוזים") הבאים :

1. קבועים – אוסף מצבים בהם העצם יכול להיות בחייו.
2. תנאים מוקדמים – תנאים שצריכים להתייחס לפני הפעלת הפונקציה.
3. תנאים מאוחרים – תנאים שחייבים להתקיים לאחר הפעלת הפונקציה.

<u>במצב אופטימלי אין תנאים מיוחדים.</u>

תכנות הגנתי (DP) - תכנות בו אני לוקח בחשבון את כל מקרי הקיצון שקיימים ומטפל בהם, העיקרון המנחה בתכנות כזה הוא שאסור להניח כלום לגבי תקינות קלט וקבלת קלט מסוים (מספרים ,מחרוזת , מערכים וכו).

MVC – מחלקה שלישית שמשלבת בין שני מחלקות המבצעות דברים שונים. לדוגמא :

מחלקה המממשת גרפיקה (UI-view) ומחלקה שמתפעלת את המשחק (model – "קוד") – ניצור מחלקה שלישית (controller) אשר תשלב בין השניים וכך יוצג המשחק על הגרפיקה.

SOLID – קווים מנחים המונעים סיבוך בכתיבת קוד.

S – single – למחלקה צריכה להיות תחום אחריות אחד – כלומר שתיהיה לה סיבה אחת להשתנות.

O – open- מחלקה צריכה להיות פתוחה להוספות וסגורה לשינויים בלי לגעת בה.

L – liskov – פונקציות המשתמשות במשתנים מסוג מחלקת אב חייבות להיות מסוגלות לפעול בצורה תקינה לגם על אובייקטים מסוג הבן מבלי להיות מודעת לסוג האובייקט בפועל.

למשל פונקציית מלבן שיורשת מריבוע תיהיה בעייתית כיוון שישנם דברים שההורשה גורמת לכך שלא יהיו דברים זהים כגון : אורך ,רוחב וכו'.

כלומר כאשר אתה יוצר מחלקת אב תיצור פונקציות כמה שיותר כלליות כך שבהורשה שלהם לא תיווצר בעיה.

תשובת מבחן : מדבר על כך שאנו יכולים להחליף אובייקטים של מחלקות שונות אשר יש ביניהם קשר של ירושה, ניתן להחליף אובייקט של מחלקת אבא באובייקט של מחלקת בן וזה מבלי לשנות את ההתנהגות של האב.

I - Interface – יש לדאוג לממשקים מצומצמים ומדויקים עד כמה שניתן כדי שהעבודה תיהיה חלקה וללא כתיבה מיותרת.

D – dependency – עצמאות - מחלקות גבוהות לא צריכות להשתמש במחלקות נמוכות.

שימוש זה נועד כדי לחלק את האובייקט לתתי נושאים והחלפה של פריט באובייקט לא תצטרך להיות מורכבת כיוון שניגש למחלקות של תתי הנושאים ושם נשנה את זה בנקודתיות. (כגון רכב וגלגלים).

מושגים :

Thread – תהליך שמריץ תוכנית , ניתן לכתוב קוד שפונים אליו כמה תהליכים במקביל ולא חייבים תוכנית אחת בכל פעם , הקוד יוצא מנקודת הנחה שיש כמה תתי תהלכים שיושבים על אותו קוד, נניח שנרצה לעשות מחלקה שמייצגת תור לגלידה ונניח שכל אדם שרוצה לקנות עומד בתור, כלומר, כל אדם מייצג תת תהליך ,אך לא ניתן לשרת 20 אנשים יחד בדיוק באותו הזמן לכן שנכתוב קוד שמותאם לטריידים וניצור איזה סינכרון בינהם, ולכן נוסיף כל מיני בדיקות כגון אם בדוגמא שלנו המוכר פנוי או תפוס , וכך אני מתעדף את התהלכים כך שכל הזמן קורה תהליך אחד ובמקביל קורים תתי תהליכים.

ממשק שנרצה לממש בכל מחלקה שנרצה להריץ על thread משלה. מייבא את הפונק' run. מבחינה טכנית- מוסיפים לחתימה של הפונק' implements runnable ומייבאים את הפונק' run.	***runnablae()***
פונק' שיוצרת תור חדש שמתווסף לתור של המעבד, לאחר מכן מפעילה על ה-*thread* את הפונק' *run* שהתקבל בבנאי.	***start()***
פונק' בוליאנית שבודקת האם ה- *thread* פעיל או שהוא מת.	**isAlive()**
אותו *thread* שהגיע לשורה זו יהיה חייב לעצור ולא להתקדם כל עוד ה- *thread* הקודם שלפניו בחיים, כלומר הוא יתקדם רק שה- *thread* שלפניו ימות.	***join()***
אותו אובייקט יעבור למצב של המתנה (יעבור ל- *"waiting pool"*) בשלב זה, כל עוד לא תופעל על האובייקט הנתון המתודה *notify*, ה- *thread* האמור ימשיך להיות במצב של המתנה .	***wait()***
מעירה את **אחד** ה- *thread* שבהמתנה על אותו אובייקט	***notify()***
מעירה את **כל** ה- *thread* שבהמתנה על אותו אובייקט	**notify All()**

מחלקה – המבנה לוגי המאגד בתוכו פונקציות ומשתני עצם שמהם נוכל ליצור אובייקטים.

אובייקט – אובייקט הוא משתנה מטיפוס המחלקה הוא נוצר ממנו ואנו בונים אובייקט עי אתחול משתני העצם של המחלקה.

מחלקה גנרית - מחלקה שלא תלויה בסוגי המשתנים. <T>

מחלקה פנימית – מחלקה בתוך מחלקה – מחלקה שהיא מוגדרת בתוך class קיים.

לא אוכל לגשת לאובייקטים מחוץ למחלקה הכללית יותר.

לדוגמא במטלה מספר 2 יצרנו NODE בתוך מחלקת GraphDS ויכולנו ליצור המון קודקודים ולגשת אליהם אך לא יכולנו לגשת וליצור NODE בGraphAlgo .

מחלקה אנונימית – מחלקה פנימית רק ללא שם. כלומר נשתמש בה רק כאשר נרצה ליצור אובייקט אחד ויחידני.

מחלקה אנונימית יכולה לרחיב מחלקה קיימת או ממשת ממשק כלומר אני לא יכול ליצור מחלקה אנונימית מאפס.

פונקציה – פונקציה לא יכולה להשתמש במשתני עצם של המחלקה ושאומרים פונקציה מתכוונים לפונקציה סטטית (פונקציה זו פעולה).

שיטה - מתודה – מופעלת על האובייקט ויש לה גישה לכל משתני העצם (פונקציה על אובייקט) פעולות שמופעלות על משני עצם של אובייקט.

אבסטרקטי – משתנה אבסטרקטי הוא משתנה שאין לי גישה לצורת המימוש שלו, כלומר המימוש מוסתר אך ניתן להשתמש בו במחלקות שונות ללא הגבלה.

משתנה פרטי – רק שיטות מהחלקה שאנו נמצאים בה כרגע יכולים לגשת אל המשתנה ולא מחוץ למחלקה.

משתנה גלובלי – ישנה גישה למשתנים ולמתודות מכל המחלקות.

משתנה סופי – משתנה שלא ניתן לגעת בערכו.

Protected – ניתן לגשת למשתנים מוגנים רק ממחלקות שיורשות ממנו מהמחלקה עצמה ולמחלקות שאיתו באותה חבילה.

דרגות חוזק והרשאות גישה -

1. **Public** – מאוד שכיח – מאפשרת גישה לכל המחלקה ולכל מקום.
2. **Protected** – לא מצוי – מאפשרת גישה למחלקות יורשות ולמחלקות באותה חבילה , לא מאפשרת גישה למחלקות חיצוניות.
3. **Private** – מאוד שכיח – לא מאפשרת גישה לאף מחלקה חיצונית ולמחלקות יורשות.

הבדל בין משתנה סטטי לעצם – משתנה סטטי משתמש בשם של המחלקה ומייבא ממנה דברים כך שכל מה שהוא סטטי הוא שייך למחלקה ומה שהוא לא סטטי הוא שייך למופע של המחלקה , כלומר עצם הוא מופע של המחלקה כלומר העצם זה סוג של ייצוג של המחלקה.

הורשה – מחלקה יורשת תקבל את כל תכונות האופי מהמחלקה המורישה (מתודות וכ'ו).

ניתן לרשת רק ממחלקה אחת כיוון שיכול להיווצר מצב בו יש שתי פונקציות אותו דבר ואז יכולה להיווצר לי שגיאת קומפלציה כלומר כיוון שבעצם ההורשה אני יורש פונקציות ממומשות זה יותר שגיאה לעומת מימוש של שני ממשקים שזה כן אפשרי מכיוון שבממשק גם אם יש את אותה פונקציה היא לא ממומשת וזה יתקמפל.

נשתמש במילה השמורה **super** – כאשר נרצה להשתמש בפונקציה של מחלקת האב והיא נדרסה במחלקת הבן כלומר כאשר יש לי שתי מתודות בעלות אותו שם גם נמצאת במחלקת האב וגם בבן היורש כאשר נפעיל את המילה super ניגש למימוש אשר נמצא במחלקת האב.

כלומר לסיכום :כאשר נרצה לקרוא להפעלתה של מתודה שהגיעה בהורשה ונדרסה על ידי הגדרה מחדש (Overriding) ומעוניינים בכך שהגרסה שתפעל היא הגרסה הישנה נשתמש במילה super .

נשתמש באופרטור "." על super ונקבל גישה למחלקת האב במחלקת הבן.

Overloading – העמסה – גאווה מאפשרת להגדיר מספר דברים בעל אותו שם (כגון משתנים או פונקציות) בתנאי שהארגומנטים או הטיפוס שונים.

לדוגמא כאשר יש שתי פונקציות בעלות אותו שם וכמות הארגומנטים שונה (מה שהפונקציה מקבלת) ניתן להעמיס על כל מתודה מס פעמים וכל עוד יש שוני מהותי בין החתימה שלה לבין המתודות האחרות שהעומסו דבר זה מתקבל אך שינוי בערך החזרה לא מספיק.

ככלל עדיף להימנע מהעמסה כיוון שלמרות שהקוד נראה אלגנטי וקצר זה מסבך מאוד את ההבנה לקורא הקוד ומקשה מאוד על שינויים במידת הצורך.

Override – דריסה - כאשר נרצה לממש מחדש מתודות אשר נמצאות במחלקת האב נדרוס אותם עי יצירת שיטה זהה בעלת אותו שם ואותם ארגומנטים והמילה override באה לתעד את דריסה זו.

כך אנחנו יכולים לשמור על החתימה של הפונקציה.

פולימורפיזם – רב צורתיות. הרעיון הוא ליצור מכנה משותף ולהתייחס לעצמים שונים בתור דברים דומים.

פעולה זו תאפשר לנו לבצע פעולות מסוימות מבלי לשנות את האובייקט עצמו.

למשל המתודה () toSting היא מתודה המשותפת לכל הבעלי חיים ולכן אני יכול ליצור מערך אשר מכיל את בעלי החיים – כלב, חתול, ציפור ולהדפיס את כולם למרות שהם אובייקטים שונים.

ממשק – מסמך שבו אנו מצהירים על השיטות ללא מימוש שלהם ולא שדות.

את הממשק נממש במחלקה ובה נממש את המתודות בעזרת השדות.

היתרון הוא בעבודה גנרית כלומר נוכל ליצור ממשק לצורה באופן כללי והמון אנשים יממשו אותו בצורה שונה למשל אחד יממש משולש והשני יממש עיגול ולכן זה מקל על העבודה.

מחלקה אבסטרקטית - מחלקה אבסטרקטית זה משהו בין ממשק לבין מחלקה רגילה – מי שיירש ממנה יקבל פונקציות ממומשות מצד אחד ומצד שני הוא גם יצטרך לממש כמה פונקציות לכן לא ניתן לייצר מופע שלה (אובייקט) כיוון שזה לא שלם ולכן לא נדע מה יקרה כאשר ניצור מופע שלה.

נשתמש בזה זה כדרך לבטא פולימורפיזם – לדוגמא, ניצור מערך של צורות (בה התכונות שונות) ונעבר על המערך הזה וכל אחד מהאיברים במערך הוא צורה שמבטאת מופע של מחלקה שיורשת מהמחלקה האבסטרקטית , כלומר, המחלקה מבטיחה שתקבל כמה פונקציות ממומשות אבל גם תממש מספר פונקציות להתאמתך ,כלומר משאירה לך את התכונות השונות למימוש שונה וכך יצרנו מכנה משותף רחב כל האפשר מבלי לכתוב מחלקה לכל צורה.

תכנות מונחה עצמים -הגדרה : מונחה עצמים זה תכנות מבוסס אובייקטים על מנת ליצור תוכנית במחשב.

העקרון של תכנות מונחה עצמים הוא בנוי על קטגוריות וכל קטגוריה מחולקת לתת קטגוריה וכל הקטגוריות משתלבות לתוכנית אחת כך שניתן לגשת לכל קטגוריה (אובייקט) ולשנות אותה בהתאם לצורך מבלי לפגוע בכל הקוד עצמו כלומר הקוד בר שינוי והשינוי מתבצע נקודתית.

Juint - מבנה אוטומטי מובנה בגאווה המאפשר לכתוב ולהריץ בדיקות בצורה נוחה ומהירה.

Git - כלי לניהול גרסאות כאשר **Github** הוא ענן לשמירת מאגרים.

פעולות שניתן לעשות דרך הגיט :

לאתחל רפוזיטורי בתקיה הנוכחית	git init
להוסיף את כל הקבצים שהשתנו מאז הקומיט האחרון אל תוך אזור הסטייג'ינג	git add --all
לבצע קומיט. כלומר, לאירוז גירסה חדשה.	git commit –m "some message"
להציג את היסטוריית הקומיטים שנעשו עד כה ברפוזיטורי הנוכחי	git log
להוסיף הפניה לרפוזיטורי המרוחק שבגיטהאב	git add remote origin YourGithubURL
להעלות את כל הרפוזיטורי שעשיתם לוקאלית אל הרפוזיטורי המרוחק (הגיטהאב שלכם)	git push origin master
להוריד את הרפוזיטורי שבענן אל המחשב שלכם	git pull origin master
להעתיק את תוכן הריפוזיורי של פרויקט כלשהו אל המחשב שלכם	git clone SomeGithubURLToCopy
"לקפוץ" לגרסא ישנה כלשהי. ניתן לראות את המספר המזהה של קומיטים ישנים ע"י פקודת הלוג.	git checkout someCommitID
הצגת השינויי בין 2 קומיטים. הקומיט הראשון צריך להיות הישן יותר.	git diff commitID commitID
יוצר ענף, שיתפצל החל מהקומיט הנוכחי עליו נמצאים בזמן הרצת הפקודה, בשם שתכתבו בפקודה עצמה.	git branch branchName
יציג את שמות כל הענפים הקיימים. כדי "לקפוץ" לענף יש להשתמש בפקודת צ'קאאוט עם שם הענף.	git branch
תצוגה גרפית של כל הקומיטים והענפים שברפוזיטורי הנוכחי	git log --graph –oneline --all

קובץ Jar – קובץ jar מתנהל כמו ספריה – כלומר לא אצטרך את כל המימוש בכדי להשתמש במה שמכיל הקובץ.

Exepcions – חריגות – התראות על מקרים חרגים בקוד.

יתבצע ע"י try ו catch והמילה השמורה throw . כאשר ה try לא מתבצע ה catch יחזיר את השגיאה ובלוק ה finally תמיד יתבצע.

ע"י המילה throw מזרוק את השגיאה עצמה – throw ("wrong") .

נשתמש בחריגות בהם ההתנהגות של הקוד אינה ברורה.

Serialize – פעולה לייצוא דאטה לוגי של האפלקציה , כלומר ייצוא אובייקטים (נקרא כדאטה)ואני נייצא אותו בפורמט מסודר ומאורגן.

לדוגמא אפליקציה צד שלישי שרוצה לקחת תמונת פרופיל וכדומה.

Deserialize – לקחת דאטה גולמי (כגון טקסטים וכדומה) ולדעת לבנות ממנו דאטה לוגי של האפליקציה.

כלומר לקחת דאטה גולמי כגון קובץ טקסט ולשחזר את האובייקט הלוגי (את הדאטה עצמה).

Json - שיטה לסידור דאטה בצורה טקסטוריאלית. כלומר, סט של חוקים איך לייצג דאטה בצורה של טקסט.

האובייקט הכי חיצוני יהיה כלוא בבלוק פקודות של סוגרים מסולסלים ובפנים יהיה שורה של ערכים של key ו value כאשר השדה של הkey חייב להיות מסוג של מחרוזת וvalue יכול להיות מכל סוג שהוא , בנוסף הvalue יכול להיות מוצג גם כאובייקט ג'ייסון.

כלומר, המשמעות היא איך אני יכול לייצג את הדאטה שלי כך שיהיה קל לתוכנות אחרות שמכירות את חוקי הפורמט להשתמש בדאטה הזו ולתרגם אותה.

גייסון זהו פורמט אחיד , שפה אוניברסלית בכדי לתקשר בין מערכות שונות בעולם וברשת.

כמובן שבעזרת גייסון ניתן גם לקרוא ולתרגם דאטה.

Serializable – זהו ממשק ריק ומנגנון שמירה והעברה של מידע בגאווה. כל מחלקה שנרצה לשמור תצטרך לממש את הממשק הריק. מנגנון זה הוא כלי מובנה בJAVA .

המנגנון עובד ע"י מיפוי של אובייקט מהזיכרון לייצוג בינארי שניתן לשמור אותו בקובץ או להעביר אותו בתקשורת. המנגנון עובד כך שהמערכת שומרת ייצוג קומפקטי של מבנה כל המחלקה שהאובייקטים שלה מרכיבים את האובייקט הנשמר.

חסרון של כלי זה הוא שהוא שייך רק לJAVA.

ההבדל בין JSON לבין Serializable - ל serializable יתרון בולט בכל שהיא למעשה אינה דורשת לכתוב קוד ופשוט "משטחת" אובייקט (מורכב ככול שיהיה) מהזיכרון לייצוג בינארי סדרתי שניתן לשמירה בקובץ או להעברה בתקשורת , הייצוג של המחלקות מהן נבנה האובייקט מאפשרות לבנות אותו מאוחר יותר (או במחשב אחר). החסרונות של השיטה כוללים את העובדה שמדובר בייצוג שמתאים רק לעבודה עם JAVA , הוא אינו קריא ע"י משתמש אנושי והוא דורש שלא יהיה שינוי במבנה המחלקות של האוביקט שנשמר.

לפיכך היתרונן של JSON כוללים : התאמה לשפות תכנות רבות ,ייצוג שאינו תלוי בגרסת קוד קונקרטית , ויכולת של משתמש אנושי לקרוא ולהבין את המידע.

: Python

הורשה - הורשה בפייתון תבצע ע"י השמת השם של מחלקת האב בסוגריים.

בנוסף בפייתון אין מימוש של ממשקים יש רק הורשות.

משתנים שלפני שמם מגיע "_" הוא משתנה פרטי.

מתודות שבמימושם דורסות :

1	**__init__ (self [,args...])** Constructor (with any optional arguments)	Sample Call : *obj = className(args)*
2	**__del__(self)** Destructor, deletes an object,	Sample Call : *del obj*
3	**__repr__(self)** Evaluable string representation,	Sample Call : *repr(obj)*
4	**__str__(self)** Printable string representation,	Sample Call : *str(obj)*
5	**__cmp__ (self, x)** Object comparison	Sample Call : *cmp(obj, x)*

טיפול בקבצים – כאשר נקרא לקובץ בפייתון נצטרך לפתוח אותו תחילה ע"י הפקודה open נקרא את הקובץ ונסגור אותו ע"י פקודת close .

לדוגמא : file object = open (file name[,access mode][,buffersize]) - פתיחה של קובץ .

W - כדי לכתוב נתונים בקובץ , R– לקריאת נתונים w ו r הם שני פקודות בaccess mode.

טיפול בחריגות excepions – בפייתון יש המון חריגות בנויות.

נשתמש ב try ו except (בגאווה השתמשנו ב try ו catch) כאשר בלוק ה try יהיה בו הקוד ו except יזרוק את ההערה.

נשתמש במילה else לאחר ה try ו except בסיטואציה בה אני ארצה להריץ קוד ללא זריקת הexcept.

המילה השמורה finally תתבצע בכל מקרה (כמו בJAVA) .

האופרטור with - משמש בטיפול בחריגים והופך את הקוד לנקי יותר לדוגמא בעת שימוש באופרטור whit אין צורך לקרוא לפונקציה לסגירת קובץ כלומר בעת שימוש באופרטור זה הוא מבטיח שחרור נכון של המשאבים.

Json – יש להשתמש במודול ה json של פייתון.

כדי להפוך מילון ל json יש להשתמש בפקודת ()json.dump .

כדי לפענח מחרוזת ב json יש להשתמש בפקודת ()json.loads .

הjson עובד כמו בjava עם key וvalue - ["name" ; "David"] .