**AMALIY MASHG‘ULOT UCHUN O‘QUV MATERIALLARI**

**1-Mavzu:** “Python dasturlash tili” faniga kirish va asosiy tushunchalari.

**10-mashg‘ulot.** Pythonda Funksiya tushunchasi. Foydalanuvchi funksiyasi..

O‘quv savollari:

1. Funksiyalarni aniqlash va uni chaqirish.
2. Parametrli va parametrsiz funksiya.
3. Anonim funksiya. Dekoratorlar. Global va lokal o‘zgaruvchi.
4. Lambda funktsiyasi.

**1. Funksiyalarni aniqlash va uni chaqirish.**

Funktsiyalar ma’lum bir vazifani bajaradigan va dasturning boshqa qismlarida qayta ishlatilishi mumkin bo‘lgan kod blokini ifodalaydi. Funktsiyalardan oldingi mavzularni tushuntirish davomida allaqachon foydalaib o‘tildi. Masalan, konsolga qiymatlarni chop etadigan **print()** funksiyasidan deyarli har bir mavzuda foydalanildi. Python ko‘plab standart o‘rnatilgan funktsiyalarga ega bo‘lgan dasturlash tilidir. Bunday standart funksiyalardan tashqari foydalanuvchi o‘ziga kerakli funktsiyalarni yaratish imkoni mavjud. Funksiya yaratish uchun quyidagi sintaksisdan foydalaniladi:

**def** function\_name([parametrlar]):

[funksiya tanasi]

Pythonda funksiya boshqa dasturlash tillaridan farqli ravishda def kalit so‘zi bilan boshlanadi. Def kalit so‘zidan keyin funksiya nomi va agar qabul qiluvchi parametrlari mavjud bo‘lsa, ularni qavs ichida, vergul bilan ajratgan holda nomlari keltiriladi va qavslardan keyin osti-usti ikki nuqta qo‘yiladi. Va keyingi qatordan funksiya bajaradigan ko‘rsatmalar bloki boshlanadi. Barcha funktsiya tanasidagi operatorlar satr boshidan ichkariga chekingan holda bir ustundan boshlanishi kerak.

Masalan, eng oddiy funktsiyaning ko‘rinishi:

**def** say\_hello():

print ("Salom")

Funktsiya nomi: say\_hello deb ataladi. U hech qanday parametrlarga ega emas va konsolga "Salom" satrini chop etadigan bitta operatorni o‘z ichiga oladi.

E’tibor bering, funktsiya iboralari funktsiya boshidan chekinishi kerak. Masalan:

**def** say\_hello():

print("Salom")

print("Xayr")

Bu yerda print("Xayr") buyrug‘i say\_hello funksiyasi bilan bir ustunda va shuning uchun bu funksiyaga tegishli bo‘lmagan operator hisoblanadi. Funktsiya tarkibiga kiruvchi blok va funktsiyaga tegishli bo‘lmagan operatorlar bir-biri bilan oldidagi qoldirilgan tab (bo‘sh joy) bilan farq qiladi.

**Funksiyaga murojaat**. Funktsiyani chaqirish uchun funktsiya nomi va uning qabul qiluvchi parametrlariga mos qiymatlar qavs ichida ko‘rsatiladi:

**funktsiya\_nomi** ([parametrlar])

Masalan, salom tekstini konsolga chiqaruvchi bitta oddiy funksiya yaratamiz va uni chaqiramiz:

**def** hello():

print ("Salom")

hello()

hello()

hello()

Bu yerda hello funksiyasi ketma-ket uch marta chaqirilmoqda. Natijada, konsolda quyidagi natijalar paydo bo‘ladi:

Salom

Salom

Salom

Shuni esda tutish kerakki, funktsiya yuqorida yaratiladi va keyin chaqiriladi.

Agar funktsiya tarkibida faqat bitta operator mavjud bo‘lsa, bu operatorni funksiya ta’rifining davomidan ikki nuqtadan keyin joylashtirilishi mumkin:

**def** hello(): print("Salom")

hello ()

Boshqa funktsiyalarni ham xuddi shunday aniqlash va chaqirish mumkin. Masalan, bir nechta funksiyalarni aniqlaymiz va bajaramiz:

**def** say\_hello():

print ("Salom")

**def** say\_goodbye():

print ("Xayr")

say\_hello ()

say\_goodbye ()

Natijaning konsolda ko‘rinishi :

Salom

Xayr

**Lokal funksiya.** Ba’zi funktsiyalarni boshqa funktsiyalar ichida aniqlash mumkin. Bunday funksiyalarni ichki funktsiyalar ya’ni ***lokal funksiyalar*** deb ataladi. Lokal funksiyalar faqat o‘zi tegishli bo‘lgan funksiya doirasida ishlatilishi mumkin. Masalan :

**def** print\_msg():

**def** say\_hello(): print("Salom")

**def** say\_bye(): print("Xayr")

say\_hello()

say\_bye()

print\_msg()

Bu yerda say\_hello() va say\_bye() funksiyalari print\_msg() funksiyasi ichida aniqlanadi va shuning uchun u uchun lokal fuksiya hisoblanadi. Shunga ko‘ra, ular faqat print\_msg() funksiyasi ichidagina ishlatilishi mumkin.

**main funksiyasi** **va undan qo‘llanilishi**. Dasturda ko‘plab funktsiyalar ishtirok etishi mumkin. Va ularning barchasini birlashtirish uchun asosiy bitta funksiya ishlatiladi. Odatda bu funksiya main deb ataladi va unda boshqa funktsiyalar chaqiriladi:

**def** main\_f ():

say\_hello ()

say\_bye ()

**def** say\_hello():

print("Salom")

**def** say\_bye():

print("Xayr")

# Asosiy funksiyani chaqirish

main\_f()

**2. Parametrli va parametrsiz funksiya**

Funksiyalarni parametrli va parametrsiz funksiyalarga ajratishimiz mumkin. Parametrsiz funksiyalar tashqaridan hech qanday qiymat qabul qilmaydi. Parametrli funksiyalar esa funksiya chaqirilayotganda biror bir qiymatni berilishini kutadigan funksiyalardir. Ushbu parametrlar orqali funksiyaga turli ma’lumotlarni uzatish mumkin. Buni **print()** funksiyasimisolida ko‘rib chiqamiz. **print()** parametr sifatida konsolga chop etilishi kerak bo‘lgan xabarli qabul qiladi.

Endi bitta oddiy, ism paramrli funksiya yaratamiz va uni chaqiramiz:

**def** say\_hello(ism):

    print(f"Salom, { ism }")

say\_hello("Jamshid")

say\_hello("Boboy")

say\_hello("Asilbek")

Say\_hello funksiyasi ism parametriga ega va funksiya chaqirilganda ushbu parametrga ba’zi qiymatlarni berish mumkin. Funksiya ichida parametrni oddiy o‘zgaruvchi sifatida ishlatish mumkin. Masalan, ushbu parametrning qiymatini print funksiyasi bilan konsolga chop etish. Shunday qilib, ifodada:

say\_hello("Jamshid ")

"Jamshid" qatori ism parametriga uzatiladi. Natijada, dasturni bajarishda quyidagi natijani konsol chiqaradi:

Salom, Jamshid

Salom, Boboy

Salom, Asilbek

Funktsiya chaqirilganda, qiymatlar pozitsiya bo‘yicha parametrlarga o‘tkaziladi. Masalan, bir nechta parametrli funksiyani quyidagicha yaratish va chaqirish mumkin:

**def** print\_per (ism, yosh):

print(f"Ism: {ism}")

print(f"Yosh: {yosh}")

print\_per ("Jamshid", 25)

Bu yerda print\_per funksiyasi ikkita parametrni qabul qiladi: ism va yosh. Funktsiyani chaqirganda :

  print\_per("Jamshid", 25)

Funksiya chaqirilish vaqtidagi unga parametr sifatida beriladigan qiymatlar mos parametrlarga qabul qilinadi. Ya’ni birinchi qiymat – "Jamshid" birinchi parametrga, ya’ni ism parametriga qabul qilinsa, ikkinchi qiymat – 25 bo‘lsa, yosh parametriga qabul qilinadi. Va funktsiya ichida parametr qiymatlari konsolga chop etiladi:

Ism: Tom

Yosh: 37

**Standart qiymatlar**

Funktsiyani yaratish mobaynida, funksiyaning mavjud parametrlari uchun boshlang‘ich qiymatlarni belgilash orqali funksiyaning ba’zi parametrlariga qiymat berishni ixtiyoriy qilinishi mumkin. Masalan :

**def** say\_hello(ism="Jamshid"):

    print(f"Salom, {ism}")

say\_hello()   # bu yerda ism = "Jamshid"

say\_hello("Bobo") # bu yerda ism = "Bobo"

Bu yerda ism parametriga qiymat berish majburiy emas. Ya’ni, agar funktsiyani chaqirishda uning ism parametriga qiymat berilmasa standart qiymat – "Jamshid" qo‘llaniladi. Ushbu dasturning konsolda ko‘rinishi quyidagicha bo‘ladi:

Salom, Jamshid

Salom, Bobo

Agar funktsiya bir nechta parametrlarga ega bo‘lsa, qiymat berish ixtiyoriy bo‘lgan parametrlar, qiymat berish majburiy bo‘lgan qilinganlardan keyin kelishi kerak. Masalan :

**def** print\_per(ism, yosh = 54):

    print(f"Ism: {ism}   Yosh: {yosh}")

print\_per("Bobo")

print\_per("Jamshid", 25)

Bu yerda yosh parametri ixtiyoriy va standart boʻyicha 54 ga teng. Undan oldin qiymat berish majburiy bo‘lgan ism parametri joylashgan. Shuning uchun funktsiyani chaqirishda yosh parametriga qiymat o‘tkaza olmaymiz, lekin ism parametriga qiymat uzatish kerak.

 Agar kerak bo‘lsa, biz barcha parametrlarni ixtiyoriy qilishimiz mumkin:

def print\_per(ism = "Jamshid", yosh = 18):

    print(f"Ism: {ism}   Yosh: {yosh}")

print\_per()  # Ism: Jamshid Yosh: 18

print\_per("Bobo")        # Ism: Bobo   Yosh: 18

print\_per("Samat", 54)   # Ism: Samat   Yosh: 54

**Nomlangan parametrlar.** Yuqoridagi misollarda funktsiya chaqirilganda parametrlarga qiymatlar joylashish o‘rni bo‘yicha uzatildi. Ammo qiymatlarni parametrlarga nomi ko‘ra uzatish ham mumkin. Buning uchun funktsiyani chaqirishda parametr nomi ko‘rsatiladi va unga qiymat beriladi:

**def** print\_per(ism, yosh):

print(f"Ism: {ism}   Yosh: {yosh}")

print\_per(yosh = 22, ism = "Jamshid")

Bunday holda, yosh va ism parametrlari nomiga ko‘ra qiymatlarni topib olishadi. Funktsiya yaratilayotganda ism parametri birinchi o‘rinda turibdi. Shunga qaramay, funktsiyani chaqirishda print\_per(yosh= 22, ism = "Jamshid") kabi chaqirilsa ham bo‘ladi va shu tariqa yosh parametriga 22 ni va ism parametriga "Jamshid" qiymatini berish mumkin.

**Funksiya parametriga faqat nomi bo‘yicha qiymat berish.** \* belgisi qaysi parametrlarga nom berilishini belgilash imkonini beradi. Ya’ni qiymatlarni faqat nomi bo‘yicha uzatish mumkin bo‘lgan parametrlar. \* belgisining o‘ng tomonida joylashgan barcha parametrlar qiymatlarni faqat nomi bilan qabul qiladi:

**def** print\_per(ism, \*, yosh, tashkilot):

print(f"Ism: {ism} Yosh: {yosh} Tashkilot: {tashkilot}")

print\_per("Bobo", yosh = 41, tashkilot = "Microsoft")

Bunday holda, yosh va tashkilot parametrlari nomi bilan qiymatlar qabul qilishadi.

Parametrlar ro‘yxatiga \* bilan prefiks qo‘yish orqali barcha parametrlarni nomlash mumkin:

**def** print\_person(\*,   ism, yosh, tashkilot):

print(f"Ism: {ism} Yosh: {yosh} Tashkilot: {tashkilot}")

Aksincha, siz qiymatlarni faqat pozitsiya bo‘yicha, ya’ni pozitsion parametrlar bo‘yicha qabul qilish mumkin bo‘lgan parametrlarni belgilashingiz kerak bo‘lsa, unda / belgisidan foydalanish mumkin.

Bunda **/** belgisidan olda joylashgan barcha parametrlar faqat joylashish o‘rniga ko‘ra qiymatlar va qabul qilishi mumkin.

**def** print\_per(ism, /, yosh, tashkilot="Micros"):

print(f"Ism:{ism} Yosh: {yosh} Tashkilot: {tashkilot}")

print\_per("Jamshid", tashkilot="ShamsWeb", yosh = 11)

print\_per("Bobo", 41)

Bunday holda, ism parametri pozitsiondir.

Bitta funktsiyada bir vaqtning o‘zida pozitsion va nomli parametrlar mavjud bo‘lishi mumkin.

**def** print\_per(ism, /,  yosh = 18, \*, tashkilot):

print(f"Ism: {ism} Yosh:{yosh} Tashkilot: {tashkilot}")

print\_per("Samat", tashkilot ="Google")

print\_per("Tomson", 37, tashkilot ="JetBrains")

print\_per("Bobo", tashkilot ="Micros", yosh= 42)

Bunday holda, ism parametri / belgisining chap tomonida joylashgan, shuning uchun u pozitsion, majburiydir hamda faqat pozitsiya bo‘yicha qiymat berilishi mumkin.

Tashkilot parametri \* belgisining o‘ng tomonida joylashganligi sababli nomlangan parametr hisoblanadi. Yosh parametri nomi bo‘yicha ham, joylashgan o‘rni bo‘yicha ham qiymat qabul qilishi mumkin.

**Aniqlanmagan sondagi parametrlar.** Funksiyaga noma’lum miqdordagi qiymatlarni uzatish uchun yulduzcha belgisidan foydalaniladi. Bu funktsiyaga nomalum miqdordagi, bir nechta qiymatlarni qabul qilishda qo‘llaniladi. Masalan, sonlar yig‘indisini hisoblash funksiyasini ko‘rib chiqamiz:

**def** sum(\*raqamlar):

natija = 0

**for** j **in** raqamlar:

natija += j

print(f"sum = {natija}")

sum(1, 2, 3, 4, 5) # sum = 15

sum(3, 4, 5, 6) # sum = 18

Bunday holda, sum funktsiyasi bitta parametrni oladi - \* raqamlar, lekin parametr nomi oldidagi yulduzcha, aslida ushbu parametr o‘rniga cheksiz miqdordagi qiymatlarni yoki qiymatlar to‘plamini qabul qilish mumkinligini ko‘rsatadi. Funktsiyaning o‘zida for siklidan foydalanib, ushbu to‘plamdan birma-bir elementlarni olish mumkin, ushbu to‘plamdan har bir qiymatni j o‘zgaruvchiga olish va u bilan qandaydir amallarni bajarish mumkin. Misol uchun, bu dasturda uzatilgan raqamlarning yig‘indisi hisoblandi.

**3. Dekoratorlar.**

Dekoratorlarni tushunish har qanday Python dasturchisi uchun muhim bosqichdir. Ushbu mavzuda dekorativlar sizga qanday qilib samaraliroq va samarali Python dasturchisi bo'lishingizga yordam berishi haqida bosqichma-bosqich qo'llanmadir.

Python-dagi dekorativlar chaqiriladigan ob'ektning o'zini doimiy ravishda o'zgartirmasdan, *chaqiriladigan* ob'ektlarning (funktsiyalar, usullar va sinflar) xatti-harakatlarini kengaytirish va o'zgartirish imkonini beradi.

Mavjud sinf yoki funktsiyaning xatti-harakatlariga "biriktirilishi" mumkin bo'lgan har qanday etarlicha umumiy funksionallik dekorativlardan foydalanishning ajoyib namunasidir. Bunga quyidagilar kiradi:

* ro'yxatga olish,
* kirish nazorati va autentifikatsiyani ta'minlash,
* vaqtni boshqarish vositalari va funktsiyalari,
* Tezlik chegarasi,
* keshlash va boshqalar.

**Nima uchun Pythonda dekorativlarni o'rganishingiz kerak?**

Bu adolatli savol. Men hozir gapirgan narsa mavhum bo'lib tuyuladi va dekorativlar Python dasturchisiga kundalik ishida qanday yordam berishini tushunish qiyin bo'lishi mumkin. Mana bir misol:

Tasavvur qiling, sizning hisobot dasturingiz biznes mantig'iga ega 30 ta funktsiyaga ega. Yomg'irli dushanba kuni ertalab menejeringiz sizga aytadi:

“TPS hisobotlarini eslaysizmi? Hisobot ishlab chiqaruvchining har bir bosqichida kirish va chiqish ma'lumotlarini jurnalga qo'shishimiz kerak. XYZ kompaniyasiga bu audit uchun kerak. Men ularga buni chorshanbagacha hal qilishimiz mumkinligini aytdim.

Python dekorativlari bilan qanchalik tanish ekanligingizga qarab, bu so'rov sizning qon bosimingizni oshiradi yoki ushbu yangi talablarni asta-sekin qabul qiladi.

Dekorator haqida ma'lumotga ega bo'lmasangiz, ehtimol siz keyingi uch kun davomida ushbu 30 ta funktsiyaning har birini o'zgartirishga va ularni qo'lda ro'yxatdan o'tish qo'ng'iroqlari bilan aralashtirib yuborishga harakat qilasiz. Umuman olganda, vaqtingizni qiziqarli o'tkazing.

Agar siz dekoratorlarni bilsangiz, xotirjam jilmayib, "Yaxshi, bugun soat 14:00 da tayyor bo'ladi" kabi bir narsa deysiz.

[Shundan so'ng, siz umumiy @audit\_log](https://habr.com/users/audit_log) dekoratorining kodini chop etasiz (faqat 10 qator uzunlikda) va uni har bir funktsiya ta'rifidan oldin qo'shasiz. Keyin siz va'da qilasiz va bir chashka qahva ichasiz.

Bu yerda men bo‘rttirib aytyapman, albatta. Ammo ozgina. Dekoratorlar haqiqatan ham kuchli bo'lishi mumkin.

Men dekorativlarni tushunish har qanday tajribali Python dasturchisi uchun muhim bosqich deb aytardim. Ular bir nechta ilg'or til tushunchalarini, shu jumladan birinchi darajali funktsiyalarning xususiyatlarini yaxshi tushunishni talab qiladi.

Lekin:

**Dekoratorlarni tushunish bunga arziydi**

Python-da dekorativlarning qanday ishlashini tushunish juda foydali.

Albatta, dekorativlarni birinchi marta taqdim etilganda tushunish juda qiyin bo'lib tuyuladi, lekin bu juda foydali xususiyat bo'lib, uni uchinchi tomon ramkalari va Python standart kutubxonasida tez-tez ko'rishingiz mumkin.

Dekoratorlarni tushuntirish har qanday yaxshi Python o'quv qo'llanmasining muhim bo'limidir. Ushbu maqolada men sizni ular bilan bosqichma-bosqich tanishtirishga harakat qilaman.

Mavzuga kirishdan oldin, keling, Python-da birinchi darajali funktsiyalarning xususiyatlarini ko'rib chiqaylik. Men [dbader.org](https://dbader.org/blog/python-first-class-functions) saytida ularga qo'llanma yozdim, uni o'qish uchun bir necha daqiqa vaqt ajratishingizni tavsiya qilaman. Bu yerda dekorativlarni tushunish uchun "birinchi darajali funktsiyalar" dan eng muhim xulosalar:

* Funktsiyalar ob'ektlardir - ular o'zgaruvchilarga tayinlanishi, boshqa funktsiyalarga o'tkazilishi va qaytarilishi mumkin.
* Funktsiyalar boshqa funktsiyalar ichida aniqlanishi mumkin va bola funktsiyasi ota-ona funksiyasining mahalliy holatini (leksik yopilishlar) olishi mumkin.

**Python dekorator asoslari**

Xo'sh, aslida dekorativlar nima? Ular boshqa funktsiyani "bezatadi" yoki "o'radi" va kodni o'ralgan funksiya bajarilishidan oldin va keyin bajarishga imkon beradi.

Dekoratorlar boshqa funktsiyalarning harakatlarini o'zgartirishi yoki kengaytirishi mumkin bo'lgan qayta ishlatiladigan modullarni aniqlash imkonini beradi. Bundan tashqari, ular sizga o'ralgan funktsiyani doimiy ravishda o'zgartirmasdan buni amalga oshirishga imkon beradi. Funktsiyaning harakati faqat bezatilganida *o'zgaradi*.

Xo'sh, oddiy dekoratorni amalga oshirish nimaga o'xshaydi? Umuman olganda, dekorator chaqiriladigan ob'ekt bo'lib, u qo'ng'iroq qilinadigan ob'ektni kiritadi va boshqa chaqiriladigan ob'ektni qaytaradi.

Quyidagi funktsiya bu xususiyatga ega va siz yozishingiz mumkin bo'lgan eng oddiy dekorativ deb hisoblanishi mumkin:

**def** null\_decorator(func):

**return** func

Ko'rib turganingizdek, null\_decorator chaqiriladigan ob'ekt bo'lib, u boshqa chaqiriladigan ob'ektni kiritish sifatida qabul qiladi va uni o'zgartirmasdan bir xil kirish ob'ektini qaytaradi.

Keling, uni boshqa funktsiyani bezash (yoki o'rash) uchun ishlatamiz:

**def** greet():

**return** 'Hello!'

greet = null\_decorator(greet)

>>> greet()

'Hello!'

Ushbu misolda greetmen funktsiyani belgilab qo'ydim va keyin uni funksiya orqali ishga tushirish orqali darhol bezadim null\_decorator. Bilaman, bu hozircha unchalik foydali ko'rinmaydi (biz maxsus null dekoratorni foydasiz qilib ishlab chiqdik, to'g'rimi?), lekin bir muncha vaqt o'tgach, bu Pythonda dekorator sintaksisi qanday ishlashini aniqroq qiladi.

O'zgaruvchini aniq chaqirish va keyin uni qayta tayinlash **null\_decorator** o'rniga, funktsiyani bir bosqichda bezash uchun Python sintaksisidan foydalanishingiz mumkin:greetgreet@

@null\_decorator

**def** greet():

**return** 'Hello!'

>>> greet()

'Hello!'

[Funksiya taʼrifidan oldin @null\_decorator](https://habr.com/users/null_decorator) satrlarini qoʻyish avval funksiyani belgilash va keyin unga dekoratorni qoʻllash bilan bir xil. Sintaksisdan foydalanish @oddiygina sintaktik shakar va bu tez-tez ishlatiladigan naqsh uchun stenografiyadir.

E'tibor bering, sintaksisdan foydalanish @funktsiyani to'g'ridan-to'g'ri belgilash vaqtida bezatadi. Bu mo'rt xakerlarsiz bezaksiz asl nusxaga kirishni qiyinlashtiradi. Shuning uchun, bezaksiz funksiyani chaqirish imkoniyatini saqlab qolish uchun siz ba'zi funktsiyalarni qo'lda bezashingiz mumkin.

Hozirgacha juda yaxshi. Keling, bu qanday amalga oshirilganini ko'rib chiqaylik.

**Dekoratorlar xatti-harakatni o'zgartirishi mumkin**

Endi siz dekorator sintaksisi bilan bir oz tanish bo'lganingizdan so'ng, keling, *aslida* biror narsani bajaradigan va bezatilgan funksiyaning harakatini o'zgartiradigan boshqa dekorator yozamiz.

Mana, bezatilgan funktsiya natijasini katta harflarga o'zgartiradigan biroz murakkabroq dekorator:

**def** uppercase(func):

**def** wrapper():

original\_result = func()

modified\_result = original\_result.upper()

**return** modified\_result

**return** wrapper

Kirish funksiyasini null dekorator qilganidek oddiygina qaytarish o‘rniga, dekorator uppercase yangi funksiyani (yopish) aniqlaydi va undan kirish funksiyasini chaqirish vaqtida uning harakatini o‘zgartirish uchun o‘rash uchun ishlatadi.

Yopish wrapperbezaksiz kirish funksiyasiga kirish huquqiga ega va kirish funksiyasi chaqirilishidan oldin va keyin qo'shimcha kodni bajarishi mumkin. (Texnik jihatdan kiritish funksiyasini umuman chaqirish shart emas).

E'tibor bering, bezatilgan funktsiya ilgari hech qachon bajarilmagan. Aslida, bu nuqtada kirish funktsiyasini chaqirish hech qanday ma'noga ega emas - dekorator chaqirilganda o'zining kiritish funktsiyasining harakatini o'zgartirishi kerak.

uppercaseDekoratorning harakatini ko'rish vaqti keldi. Agar asl funktsiyani u bilan bezasangiz nima bo'ladi greet?

@uppercase

**def** greet():

**return** 'Hello!'

>>> greet()

'HELLO!'

Umid qilamanki, bu siz kutgan natija bo'ldi. Keling, bu yerda nima sodir bo'lganini batafsil ko'rib chiqaylik. null\_decorator dan farqli o'laroq, dekorator funksiyani bezashda *boshqa funksiya ob'ektini* uppercase qaytaradi :

>>> greet

<function greet at 0x10e9f0950>

>>> null\_decorator(greet)

<function greet at 0x10e9f0950>

>>> uppercase(greet)

<function uppercase.<locals>.wrapper at 0x10da02f28>

Avval ko'rganingizdek, bu bezatilgan funksiya chaqirilganda uning harakatini o'zgartirish uchun kerak. Katta harf dekoratorining o'zi funksiyadir. Va u bezatgan kirish funktsiyasining "kelajakdagi xatti-harakatiga" ta'sir qilishning yagona yo'li - kirish funktsiyasini yopish bilan almashtirish (yoki o'rash).

Shuning uchun uppercase u keyinroq chaqirilishi mumkin bo'lgan boshqa funktsiyani (yopish) belgilaydi va qaytaradi, dastlabki kiritish funktsiyasini ishga tushiradi va uning chiqishini o'zgartiradi.

Dekoratorlar qo'ng'iroq qilinadigan ob'ektning harakatini o'ram yordamida o'zgartiradilar, shuning uchun siz doimo asl nusxani o'zgartirishingiz shart emas. Chaqiriladigan ob'ekt doimiy o'zgarishlarga duch kelmaydi - uning xatti-harakati faqat bezatilganida o'zgaradi.

Bu sizga mavjud funktsiyalar va sinflarni jurnallar va boshqa vositalar kabi qayta foydalanish mumkin bo'lgan modullar bilan "biriktirish" imkonini beradi. Bu dekorativlarni Python-da ko'pincha standart kutubxona va uchinchi tomon paketlarida ishlatiladigan kuchli xususiyatga aylantiradi.

Funktsiyaga bir nechta dekorator qo'llanilishi ajablanarli emas. Bu ularning effektlarini to'playdi va dekorativlarni qayta foydalanish mumkin bo'lgan modullar kabi foydali qiladi.

Mana bir misol. Keyingi ikkita dekorator bezatilgan funksiyaning chiqish satrini HTML teglari bilan o‘rab oladi. Teglar qanday joylashtirilganiga qarab, Python bir nechta dekorativlarni qo'llash tartibini ko'rishingiz mumkin:

**def** strong(func):

**def** wrapper():

**return** '<strong>' + func() + '</strong>'

**return** wrapper

**def** emphasis(func):

**def** wrapper():

**return** '<em>' + func() + '</em>'

**return** wrapper

Keling, ushbu ikkita dekoratorni olib, ularni greetbir vaqtning o'zida bizning funktsiyamizga qo'llaymiz. Buni amalga oshirish uchun siz odatiy sintaksisdan foydalanishingiz @va bitta funktsiya ustiga bir nechta dekorativlarni joylashtirishingiz mumkin:

@strong

@emphasis

**def** greet():

**return** 'Hello!'

Bezatilgan funktsiyani ishga tushirsangiz, qanday natija ko'rishni kutasiz? [@emphasis](https://habr.com/users/emphasis) dekoratori avval o'z tegini qo'shadimi yoki <em>@strong [ustunlik](https://habr.com/users/strong)   qiladimi? Bezatilgan funktsiyani chaqirganingizda nima sodir bo'ladi:

>>> greet()

'<strong><em>Hello!</em></strong>'

Bu erda siz dekorativlar qanday tartibda ishlatilganligini aniq ko'rishingiz mumkin: *pastdan yuqoriga*. Dastlab kiritish funksiyasi [@emphasis](https://habr.com/users/emphasis) decorator bilan o'ralgan, so'ngra hosil bo'lgan (bezatilgan) funksiya yana [@strong](https://habr.com/users/strong) decorator bilan o'ralgan .

Ushbu pastdan yuqoriga tartibni eslash uchun men bu xatti-harakatni "dekorator to'plami" deb atashni yaxshi ko'raman. Siz stackni pastki qismdan qurishni boshlaysiz va keyin yuqoriga ko'tarilish uchun yangi bloklarni qo'shishda davom etasiz.

Agar biz yuqoridagi misolni buzsak va dekorativlarni qo'llash uchun sintaksisdan foydalanmasak @, dekorativ funktsiyalarga qo'ng'iroqlar zanjiri quyidagicha ko'rinadi:

decorated\_greet = strong(emphasis(greet))

Bu yerda yana dekoratorning birinchi qo'llanilishini ko'rishingiz mumkin emphasis, keyin esa natijada o'ralgan funksiya yana dekoratorga o'ralgan strong.

Bu shuningdek, dekorativ qoplamaning chuqur darajalari oxir-oqibat ishlashga ta'sir qiladi, chunki ular ichki funksiya chaqiruvlarini qo'shishda davom etadilar. Amalda bu odatda muammo emas, lekin agar siz unumdorlikni talab qiluvchi kod ustida ishlayotgan bo'lsangiz, buni yodda tutish kerak.

Argumentlarni qabul qiladigan dekoratsiya funktsiyalari

greetHozirgacha berilgan barcha misollar hech qanday dalil talab qilmaydigan oddiy nullary funktsiyani bezatadi. Shunday qilib, siz bu erda ko'rgan dekoratorlar kiritish funktsiyasiga argumentlarni uzatish bilan shug'ullanmagan.

Agar siz ushbu dekoratorlardan birini argumentlarni qabul qiluvchi funktsiyaga qo'llamoqchi bo'lsangiz, u to'g'ri ishlamaydi. O'zboshimchalik bilan argumentlar oladigan funktsiyani qanday bezash kerak?

Bu yerda Python funksiyasi o'zgaruvchan sonli argumentlar bilan ishlashda \*args*yordam* beradi. \*\*kwargs Dekorator proxy ushbu xususiyatdan foydalanadi:

**def** proxy(func):

**def** wrapper(\*args, \*\*kwargs):

**return** func(\*args, \*\*kwargs)

**return** wrapper

Ushbu dekorativning ikkita diqqatga sazovor tomoni bor:

* U barcha pozitsion va kalit argumentlarni yig'ish va ularni o'zgaruvchilarda(va) saqlash uchun yopish ta'rifidagi \*va operatorlaridan foydalanadi .\*\*wrapperargskwargs
* \*Keyin oʻramni yopish “argumentni ochish” va yordamida toʻplangan argumentlarni asl kiritish funksiyasiga oʻtkazadi \*\*.

(Yulduzcha va qoʻsh yulduzcha operatorlarining maʼnosi haddan tashqari yuklanganligi va ular qoʻllanilgan kontekstga qarab oʻzgarib turishi biroz uyatli. Lekin bu fikrni tushunasiz degan umiddaman).

Keling, dekorativning orqasidagi texnikani proxyfoydaliroq amaliy misol bilan kengaytiraylik. Mana, tracefunksiyaning argumentlarini va ishlayotgan natijalarini chop etadigan dekorator:

**def** trace(func):

**def** wrapper(\*args, \*\*kwargs):

print(f'TRACE: calling {func.\_\_name\_\_}() '

f'with {args}, {kwargs}')

original\_result = func(\*args, \*\*kwargs)

print(f'TRACE: {func.\_\_name\_\_}() '

f'returned {original\_result!r}')

**return** original\_result

**return** wrapper

Funksiyani bezash traceva uni chaqirish orqali siz bezatilgan funksiyaga uzatilgan argumentlarni va uning qaytish qiymatini chop etishingiz mumkin. Bu hali ham o'yinchoq misolidir, ammo bu nosozliklarni tuzatish uchun ajoyib yordam bo'ladi:

@trace

**def** say(name, line):

**return** f'{name}: {line}'

>>> say('Jane', 'Hello, World')

'TRACE: calling say() with ("Jane", "Hello, World"), {}'

'TRACE: say() returned "Jane: Hello, World"'

'Jane: Hello, World'

Nosozliklarni tuzatish haqida gapiradigan bo'lsak, dekorativlarni tuzatishda bir nechta narsalarni yodda tutish kerak.

"Debuggable" dekorativlarni qanday yozish kerak

Dekoratordan foydalansangiz, aslida bir funktsiyani boshqasiga almashtirasiz. Bu jarayonning kamchiliklaridan biri shundaki, u asl (bezaksiz) funksiyaga biriktirilgan metama’lumotlarning bir qismini “yashiradi”.

Masalan, asl funktsiya nomi, uning docstring va parametrlar ro'yxati yopish orqali yashiriladi:

**def** greet():

"""Return a friendly greeting."""

**return** 'Hello!'

decorated\_greet = uppercase(greet)

Agar siz ushbu funksiyaning istalgan metamaʼlumotlariga kirishga harakat qilsangiz, uning oʻrniga oʻramning yopish metamaʼlumotlarini koʻrasiz:

>>> greet.\_\_name\_\_

'greet'

>>> greet.\_\_doc\_\_

'Return a friendly greeting.'

>>> decorated\_greet.\_\_name\_\_

'wrapper'

>>> decorated\_greet.\_\_doc\_\_

**None**

Bu nosozliklarni tuzatish va Python tarjimoni bilan ishlashni noqulay va qiyinlashtiradi. Yaxshiyamki, buning tezkor yechimi bor: Python standart kutubxonasiga kiritilgan [functools.wraps dekoratori.](https://docs.python.org/3/library/functools.html" \l "functools.wraps)

functools.wrapsSiz o'zingizning dekoratorlaringizda dekoratsiya qilinmagan funksiyadan bezatuvchining yopilishiga yetim metama'lumotlarni nusxalash uchun foydalanishingiz mumkin . Mana bir misol:

**import** functools

**def** uppercase(func):

@functools.wraps(func)

**def** wrapper():

**return** func().upper()

**return** wrapper

Dekorator tomonidan qaytarilgan oʻramni yopishni qoʻllash functools.wrapshujjat qatori va kiritish funksiyasining boshqa metamaʼlumotlarini oʻrab oladi:

@uppercase

**def** greet():

"""Return a friendly greeting."""

**return** 'Hello!'

>>> greet.\_\_name\_\_

'greet'

>>> greet.\_\_doc\_\_

'Return a friendly greeting.'

functools.wrapsO'zingiz yozgan barcha dekorativlardan foydalanishni tavsiya qilaman . Bu ko'p vaqtni talab qilmaydi va kelajakda disk raskadrovka paytida sizni (va boshqalarni) ko'p bosh og'rig'idan xalos qiladi.

**4. Lambda funktsiyasi**

Python dasturlash tilidagi lambda ifodalari - bu lambda operatori yordamida aniqlangan kichik anonim funksiyalar. Lambda ifodasining sintaksisi quyidagicha:

**lambda** [parametrlar]: operatorlar

Eng oddiy lambda ifodasini aniqlaymiz:

msg = **lambda**: print ("salom")

msg()    # salom

Bu yerda lambda ifodasi msg o‘zgaruvchisiga yuklatilgan. Ushbu misolda lambda ifodasi hech qanday parametrga ega emas, hech narsa qaytarmaydi va shunchaki konsolga "salom" xabarini chop etadi. Va msg o‘zgaruvchisi orqali bu lambda ifodasini oddiy funktsiya kabi chaqirishimiz mumkin. Aslida, u quyidagi funktsiyaga o‘xshaydi:

**def** msg():

    print("salom")

Agar lambda ifodasi parametrlarga ega bo‘lsa, ular lambda kalit so‘zidan keyin aniqlanadi. Agar lambda ifodasi natijani qaytarsa, u holda u ikki nuqtadan keyin ko‘rsatiladi. Masalan, sonning kvadratini qaytaruvchi lambda ifodasini yozaylik:

kvadrat = **lambda** a : a \* a

print(kvadrat(4))     # 16

print(kvadrat(5))     # 25

Bunday holda, lambda ifodasi bitta a nomli parametrni qabul qiladi va undan keyingi Ikki nuqtaning o‘ng tomonida qaytariladigan qiymat - a \* a. Ushbu lambda ifodasi quyidagi funktsiyaning alternativi hisoblanadi:

**def** kvadrat2(a): **return** a \* n

Xuddi shu usulda, bir nechta parametrlarni qabul qiluvchi lambda ifodalarini yaratish mumkin:

sum = **lambda** a, b: a + b

print(sum(4, 5))     # 9

print(sum(5, 6))     # 11

Lambda ifodalari funksiya ta’riflarini biroz qisqartirishga imkon bersada, ular faqat bitta operatsiyani bajarishi mumkinligi bilan cheklangan. Biroq, parametr sifatida o‘tish yoki boshqa funktsiyaga qaytish uchun funktsiyadan foydalanish kerak bo‘lgan hollarda ular juda qulay bo‘lishi mumkin. Masalan, lambda ifodasini parametr sifatida o‘tkazish:

**def** do\_operatsiya(a, b, operatsiya):

    result = operatsiya(a, b)

    print(f"result = {result}")

do\_operatsiya(5, 4, **lambda** a, b: a + b)  #result = 9

do\_operatsiya(5, 4, **lambda** a, b: a \* b)  #result = 20

Bunday holda, oxirgi mavzuda bo‘lgani kabi, ularni parametr sifatida jo‘natish uchun funktsiyalarni belgilashimiz shart emas.

Xuddi shu narsa funktsiyalardan lambda ifodalarini qaytarish uchun ham amal qiladi:

**def** tanlash\_funk(tanlangan):

**if** tanlangan == 1:

**return** **lambda** a, b: a + b

**elif** tanlangan == 2:

**return** **lambda** a, b: a - b

**else**:

**return** **lambda** a, b: a \* b

operation = tanlash\_funk (1) # operation = a+b

print(operation(10, 6))  # 16

operation = tanlash\_funk (2) # operation = a-b

print(operation(10, 6))  # 4

operation = tanlash\_funk (3) # operation = a\*b

print(operation(10, 6))  # 60

**Nazorat savollari:**

1. Lambda funksiyasining vazifasi nima?
2. Lambda funksiyasining sintaksisi va ishlash prinsipi qanday?
3. Oddiy funksiyadan qanday afzallik va kamchiliklarga ega?
4. Lokal funksiya deb qanday funksiyalarga aytiladi?
5. Funksiyalar dasturlashda qanday ro‘l o‘ynaydi?
6. Foydalanuvchi funksiyalari qanday yaratiladi?
7. Funksiya tanasi va funksiyaga tegishli bo‘lmagan operatorlar qanday farqlanadi?