GİRİŞ

Tahmin edebileceğiniz gibi Python (C, C++, Perl, Ruby ve benzerleri gibi) bir programlama dilidir ve tıpkı öteki programlama dilleri gibi, önünüzde duran kara kutuya, yani bilgisayara hükmetmenizi sağlar.

Guido Van Rossum adlı Hollandalı bir programcı tarafından 90'lı yılların başında geliştirilmeye başlanmıştır. Çoğu insan, isminin Python olmasına aldanarak, bu programlama dilinin, adını piton yılanından aldığını düşünür. Ancak zannedildiğinin aksine bu programlama dilinin adı piton yılanından gelmez. Guido Van Rossum bu programlama dilini, The Monty Python adlı bir İngiliz komedi grubu Monty Python's Flying Circus adlı gösterisinden esinlenerek adlandırmıştır. Ancak her ne kadar gerçek böyle olsa da, Python programlama dilinin pek çok yerde bir yılan figürü ile temsil edilmesi neredeyse bir gelenek halini almıştır.

Python nasıl telafuz edilir?

Python programlama dili üzerine bu kadar söz söyledik. Peki yabancı bir kelime olan Python'ı nasıl telaffuz edeceğimizi biliyor muyuz?

Geliştiricisi Hollandalı olsa da python İngilizce bir kelimedir. Dolayısıyla bu kelimenin telaffuzunda İngilizcenin kuralları geçerli. Ancak bu kelimeyi hakkıyla telaffuz etmek, ana dili Türkçe olanlar için pek kolay değil. Çünkü bu kelime içinde, Türkçede yer almayan ve telaffuzu peltek s'yi andıran [th] sesi var. İngilizce bilenler bu sesi think (düşünmek) kelimesinden hatırlayacaklardır. Ana dili Türkçe olanlar think kelimesini genellikle [tink] şeklinde telaffuz eder. Dolayısıyla python kelimesini de [paytın] şeklinde telaffuz edebilirsiniz.

Python kelimesini tamamen Türkçeleştirerek [piton] şeklinde telaffuz etmeyi yeğleyenler de var. Elbette siz de dilinizin döndüğü bir telaffuzu tercih etmekte özgürsünüz.

Bu arada, eğer python kelimesinin İngilizce telaffuzunu dinlemek istiyorsanız howjsay.com adresini ziyaret edebilir, Guido Van Rossum'un bu kelimeyi nasıl telaffuz ettiğini merak ediyorsanız da https://www.youtube.com/watch?v=UIDdgeISLUI adresindeki tanıtım videosunu izleyebilirsiniz.

Platform Desteği

Python programlama dili pek çok farklı işletim sistemi ve platform üzerinde çalışabilir. GNU/Linux, Windows, Mac OS X, iOS ve Android gibi, belki adını dahi duymadığınız pek çok ortamda Python uygulamaları geliştirebilirsiniz. Ayrıca herhangi bir ortamda yazdığınız bir Python programı, üzerinde hiçbir değişiklik yapılmadan veya ufak değişikliklerle başka ortamlarda da çalıştırılabilir

Neden Python?

Python programlarının en büyük özelliklerinden birisi, C ve C++ gibi dillerin aksine, derlenmeye (compile) gerek olmadan çalıştırılabilmeleridir. Python'da derleme işlemi ortadan kaldırıldığı için, bu dille oldukça hızlı bir şekilde program geliştirilebilir.

Python programlama dilinin basit ve temiz söz dizimi, onu pek çok programcı tarafından tercih edilen bir dil haline getirmiştir. Python'ın söz diziminin temiz ve basit olması sayesinde hem program yazmak, hem de başkası tarafından yazılmış bir programı okumak, başka dillere kıyasla çok kolaydır.

Python'ın yukarıda sayılan özellikleri sayesinde dünya çapında ün sahibi büyük kuruluşlar (Google, YouTube ve Yahoo! gibi) bünyelerinde her zaman Python programcılarına ihtiyaç duyuyor.

Python programlama dili ve bu dili hakkıyla bilenler sadece uluslararası şirketlerin ilgisini çekmekle kalmıyor. Python son zamanlarda Türkiye'deki kurum ve kuruluşların da dikkatini çekmeye başladı. Bu dil artık yavaş yavaş Türkiye'deki üniversitelerin müfredatında da kendine yer buluyor.

Python'u Yükleme ve Kurulum

https://www.python.org/downloads/

Herkes Python'u farklı uygulamalar için kullandığından, Python'u ve gerekli eklenti paketlerini kurmak için tek bir çözüm yoktur. Ücretsiz Anaconda dağıtımını kullanmanızı öneririm: https://www.anaconda.com/

Windows işletim sisteminiz varsa, Python kurulumu gerçekleştirmek için Anaconda yükleyicisini indirin ve adımları takip edin: https://www.anaconda.com/products/individual

Benzer şekilde Linux ve MacOS işletim sistemine sahip makineleriniz için de yükleme gerçekleştirebilirsiniz.

Python 2 ve Python 3

Her programlama dili, yeni fikirler ve teknolojiler ortaya çıktıkça gelişir ve Python geliştiricileri, dili sürekli olarak daha çok yönlü (versatile) ve güçlü hale getirmektedir.

Python 3.x yorumlayıcı serisinin ilk versiyonu 2008'in sonunda yayınlandı. Daha önce yazılmış bazı Python 2.x kodlarını uyumsuz hale getiren bir dizi değişiklik içeriyordu. Python'un Guido van Rossum tarafından yazılan 1991'deki ilk sürümünden bu yana 17 yıl geçtiği için, Python 3'ün "kırılma" sürümünü oluşturmanın, bu süre zarfında öğrenilen dersler göz önüne alındığında daha iyi olduğu düşünülüyordu.

2012'de, birçok paket tam olarak Python 3 uyumlu hale getirilmediğinden, bilim ve veri analizi topluluğunun çoğu hala Python 2.x kullanıyordu. Ancak Python 2.x (ve kritik güvenlik yamaları) 2020 yılında geliştirme ömrünün sonuna ulaştı ve bu nedenle Python 2 kullanarak yeni projeler başlatmak artık iyi bir fikir değil.

Bu nedenle, bu eğitimde, yaygın olarak dağıtılan, iyi desteklenen kararlı bir sürüm olan Python 3'ü kullanacağız.

Hangi Python'u kullanıyorum?

Python versiyonunuzu kontrol etmek için komut satırınızda (command line - Windows), terminalde (Mac ve Linux/Ubuntu) `python --version` komutunu çalıştırınız.

Betiğinizde (script) Python versiyonunuzu kontrol etmek için, modülü içeri almak (import) üzere `import sys` komutunu çalıştırınız ve kodunuzdaki ayrıntılı sürüm bilgilerini bulmak için `sys.version` , `sys.executable` , veya `sys.version_info` 'u kullanınız

In [ ]:
```
import sys
print(sys.executable, '\n')
print(sys.version, '\n')
print(sys.version_info, '\n')
```

```
/usr/local/opt/python@3.10/bin/python3.10

3.10.8 (main, Oct 13 2022, 10:17:43) [Clang 14.0.0 (clang-1400.0.29.102)]

sys.version_info(major=3, minor=10, micro=8, releaselevel='final', serial=0)
```

Python Paketlerini Yükleme veya Güncelleme

Eğitim esnasında, Anaconda dağıtımına dahil olmayan ek Python paketleri kurmak isteyebilirsiniz. Genel olarak, bunlar aşağıdaki komutla yüklenebilir:

```
conda install paketin_ismi
```

Bu işe yaramazsa, paketi `pip` paket yönetim aracını kullanarak da kurabilirsiniz (Python3 için `pip3` kullanabilirsiniz):

```
pip install paketin_ismi
```

`conda update` komutunu kullanarak paketleri güncelleyebilirsiniz (update):

```
conda update paketin_ismi
```

pip ayrıca `--upgrade` bayrağını (flag) kullanan yükseltmeleri (upgrade) de destekler:

```
pip install --upgrade paketin_ismi
```

Python Toplulukları ve Sorunlarınıza Cevap Bulma: Stackoverflow.com ve Stackexchange.com

https://stackoverflow.com/

https://stackexchange.com/

- Matematik - https://math.stackexchange.com/
- İstatistik - https://stats.stackexchange.com/
- Yazılım Mühendisliği - https://softwareengineering.stackexchange.com/
- Veri Bilimi - https://datascience.stackexchange.com/
- TeX / LaTeX - https://tex.stackexchange.com/
- Sistem ve Ağ Yönetimi - https://serverfault.com/
- Veri Tabanı Yönetimi - https://dba.stackexchange.com/
- Web Uygulamaları - https://webapps.stackexchange.com/
- Unix / Linux - https://unix.stackexchange.com/

Temel Python Kütüphanelerinin Tanıtımı: Numpy, pandas, matplotlib, SciPy, scikit-learn ve statsmodels

Python veri ekosistemine ve eğitim boyunca kullanılan kütüphanelere daha az aşina olanlar için, bazı önemli kütüphanelere kısa bir genel bakış sunacağım.

NumPy

Numerical Python'un kısaltması olan NumPy, Python'da uzun süredir sayısal hesaplamanın temel taşı olmuştur. Python'da sayısal verileri içeren çoğu bilimsel uygulama için gereken veri yapılarını, algoritmaları ve kütüphanelerini sağlar. NumPy, diğer şeylerin yanı sıra şunları içerir:

- Hızlı ve verimli bir çok boyutlu dizi (array) nesnesi `ndarray`
- Dizilerle eleman bazında (element-wise) hesaplamalar veya diziler arasında matematiksel işlemler gerçekleştirmek için fonksiyonlar
- Dizi tabanlı veri kümelerini diske okumak ve yazmak için araçlar
- Doğrusal cebir işlemleri, Fourier dönüşümü ve rastgele sayı üretimi
- Python uzantılarının ve yerel C veya C++ kodunun NumPy'nin veri yapılarına ve hesaplama tesislerine erişmesini sağlayan olgun bir C API'si

NumPy'nin Python'a eklediği hızlı dizi işleme yeteneklerinin ötesinde, veri analizindeki birincil kullanımlarından biri, algoritmalar ve kütüphaneler arasında veri aktarımı için bir konteyner

olmasıdır. Sayısal veriler için, NumPy dizileri, verileri depolamak ve işlemek için diğer yerleşik Python veri yapılarından daha verimlidir.

Ayrıca, C veya Fortran gibi daha düşük düzeyli bir dilde yazılmış kütüphaneler, verileri başka bir bellek temsiline kopyalamadan NumPy dizisinde depolanan veriler üzerinde çalışabilir. Bu nedenle, Python için birçok sayısal hesaplama aracı, NumPy dizilerini birincil veri yapısı olarak kabul eder veya NumPy ile sorunsuz birlikte çalışabilirliği hedefler.

Pandas

pandas, yapılandırılmış veya tablo halindeki verilerle çalışmayı hızlı, kolay ve anlamlı hale getirmek için tasarlanmış üst düzey veri yapıları ve fonksiyonlar sağlar.

Bu eğitimde kullanılacak olan pandas kütüphanesindeki birincil nesneler, hem satır hem de sütun etiketlerine sahip tablo şeklinde, sütun yönelimli (column-oriented) bir veri yapısı olan `DataFrame`, ve tek boyutlu etiketli bir dizi nesnesi olan `Series`'dir.

pandas, NumPy'nin yüksek performanslı, dizi hesaplama fikirlerini elektronik tabloların ve ilişkisel veritabanlarının (SQL gibi) esnek veri işleme yetenekleriyle harmanlar.

Yeniden şekillendirmeyi (reshaping), dilimlemeyi (slicing), toplamaları (aggregation) gerçekleştirmeyi ve veri alt kümelerini seçmeyi kolaylaştırmak için gelişmiş indeksleme işlevselliği sağlar.

Veri işleme, hazırlama ve temizleme, veri analizinde çok önemli bir beceri olduğundan, pandas kütüphanesi bu eğitimin ana odak noktalarından biridir.

matplotlib

matplotlib, grafikler ve diğer iki boyutlu veri görselleştirmeleri üretmek için en popüler Python kütüphanesidir. Başlangıçta John D. Hunter tarafından yaratıldı ve şimdi büyük bir geliştirici ekibi tarafından sürdürülüyor.

Yayınlanmak üzere uygun grafikler oluşturmak için tasarlanmıştır. Python programcılarının kullanabileceği başka görselleştirme kütüphaneleri olsa da (mesela seaborn ve plotly kütüphaneleri), matplotlib en yaygın kullanılanıdır ve bu nedenle ekosistemin geri kalanıyla genellikle iyi bir entegrasyona sahiptir. Varsayılan bir görselleştirme aracı olarak güvenli bir seçimdir.

SciPy

SciPy, bilimsel hesaplamada bir dizi farklı standart problem alanını ele alan bir paketler topluluğudur. İşte dahil olan paketlerin bir örneği:

- **scipy.integrate**: Sayısal entegrasyon rutinleri ve diferansiyel denklem çözücüler

- **scipy.linalg** `numpy.linalg` modülü içinde sağlananların ötesine geçen doğrusal cebir (Linear algebra) rutinleri ve matris ayrıştırmaları (matrix decomposition)
- **scipy.optimize**: Fonksiyon optimize ediciler (küçültücüler - minimize ediciler) ve kök bulma algoritmaları
- **scipy.signal**: sinyal işleme araçları
- **scipy.sparse**: seyrek (sparse) matrisler ve seyrek lineer sistem çözücüler
- **scipy.special**: Gama fonksiyonu gibi birçok yaygın matematiksel fonksiyonu uygulayan bir Fortran kitaplığı olan SPECFUN'un etrafındaki sarmalayıcı (wrapper)
- **scipy.stats**: standart sürekli ve kesikli olasılık dağılımları (yoğunluk fonksiyonları, örnekleyiciler, dağılım fonksiyonları v.b.), çeşitli istatistiksel testler ve daha tanımlayıcı istatistikler

NumPy ve SciPy birlikte birçok geleneksel bilimsel hesaplama uygulaması için makul ölçüde eksiksiz ve olgun bir hesaplama temeli oluşturur.

scikit-learn

Projenin 2010'daki başlangıcından bu yana, scikit-learn, Python programcıları için önde gelen genel amaçlı makine öğrenimi araç takımı haline geldi. Sadece yedi yılda, dünyanın dört bir yanından 1.500'den fazla kişi projeye katkıda bulundu. Aşağıdaki gibi modeller için alt modüller içerir:

- Sınıflandırma: DVM, en yakın komşular, rastgele orman, lojistik regresyon, vb.
- Regresyon: Lasso regresyon, Ridge regresyonu vb.
- Kümeleme: k-ortalamalar, spektral kümeleme, vb.
- Boyut azaltma: Temel Bileşenler Analizi, öznitelik (değişken) seçimi, matris çarpanlara ayırma, vb.
- Model seçimi: Izgara arama, çapraz doğrulama, metrikler
- Ön işleme: Öznitelik çıkarma, normalleştirme vb.

pandas, statsmodels ve Jupyter ile birlikte scikit-learn kütüphanesi, Python'un üretken bir veri bilimi programlama dili olmasını sağlamak için kritik öneme sahiptir.

statsmodels

statsmodels, R programlama dilinde popüler olan bir dizi regresyon analizi modelini uygulayan Stanford Üniversitesi istatistik profesörü Jonathan Taylor'ın çalışmasıyla tohumlanan bir istatistiksel analiz paketidir. Skipper Seabold ve Josef Perktold, 2010 yılında resmi olarak yeni statsmodels projesini oluşturdular ve o zamandan beri projeyi, ilgili kullanıcılar ve katkıda bulunanlardan oluşan kritik bir kitleye dönüştürdüler. Nathaniel Smith, R'nin formül sisteminden esinlenerek statsmodels için bir formül veya model spesifikasyon çerçevesi sağlayan Patsy projesini geliştirdi.

Scikit-learn ile karşılaştırıldığında, statsmodels klasik istatistik ve ekonometri için algoritmalar içerir. Aşağıdaki gibi alt modülleri içerir:

- **Regresyon modelleri**: Doğrusal regresyon, genelleştirilmiş doğrusal modeller (generalized linear models), sağlam doğrusal modeller (robust linear models), doğrusal karışık etki modelleri, vb.
- Varyans Analizi (ANOVA)
- **Zaman serileri analizi**: AR, ARMA, ARIMA, VAR ve diğer modeller
- **Parametrik olmayan yöntemler**: Çekirdek (kernel) yoğunluğu tahmini, çekirdek regresyonu
- İstatistiksel model sonuçlarının görselleştirilmesi

statsmodels kütüphanesi, parametreler için belirsizlik tahminleri ve p-değerleri sağlayarak istatistiksel çıkarsamaya daha fazla odaklanır. scikit-learn ise aksine daha çok tahmin odaklıdır.

Python'un Zen'i

Deneyimli Python programcıları, sizi karmaşıklıktan kaçınmaya ve mümkün olduğunca basitliği hedeflemeye teşvik edecektir. Python topluluğunun felsefesi Tim Peters tarafından yazılan "The Zen of Python"da yer almaktadır. İyi Python kodu yazmak için bu kısa ilkeler dizisine, yorumlayıcınıza `import this` yazarak erişebilirsiniz.

Python için PEP20 (https://peps.python.org/pep-0020/) Python'un Zen'ini de vermektedir.

In [2]:
```
import this
```

```
The Zen of Python, by Tim Peters

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

Güzel, çirkinden iyidir.
Açık, dolaylı olmaktan daha iyidir.
Basit, komplike daha iyidir.
Komplike, anlaşılmaz olmaktan iyidir.
Düz, iç içe geçmiş olmaktan daha iyidir.
Seyrek, yoğundan daha iyidir.
Okunabilirlik önemlidir.
Özel durumlar kuralları çiğneyecek kadar özel değildir.
Her ne kadar pratiklik saflığı yense de.
Hatalar asla sessizce el altı edilmemelidir.
Açıkça susturulmadıkça.
Belirsizlik karşısında, tahmin etme gücünü kullanmanın sizi baştan çıkarmasına izin vermeyin.
Bunu yapmanın bir - ve tercihen yalnızca bir - bariz yolu olmalıdır.
Her ne kadar Hollandalı değilseniz, bu yol ilk başta açık olmayabilir.
Şimdi hiç olmamasından iyidir.
Her ne kadar "hiçbir zaman" asla "şimdi"den daha iyi olmasa da.
Uygulamayı açıklamak zorsa, bu kötü bir fikirdir.
Uygulamayı açıklamak kolaysa, bu iyi bir fikir olabilir.
Ad alanları, harika bir fikirdir - hadi bunlardan daha fazlasını yapalım!

PEP8 – Python Stillendirme Kılavuzu

Birisi Python dilinde bir değişiklik yapmak istediğinde, bir Python Geliştirme Önerisi (Python Enhancement Proposal - PEP) yazar. En eski PEP'lerden biri, Python programcılarına kodlarını nasıl şekillendirecekleri konusunda talimat veren PEP 8'dir. PEP 8 oldukça uzundur, ancak çoğu bu eğitimde göreceklerinizden daha karmaşık kodlama yapılarıyla ilgilidir.

https://www.python.org/dev/peps/pep-0008/

Python stil kılavuzu, kodun yazıldığından daha sık okunduğu anlayışıyla yazılmıştır. Kodunuzu bir kez yazacak ve ardından hata ayıklamaya (debugging) başladığınızda okumaya başlayacaksınız. Bir programa yeni özellikler (features) eklediğinizde, kodunuzu okumak için daha fazla zaman harcarsınız. Kodunuzu diğer programcılarla paylaştığınız zaman onlar da kodunuzu okuyacaktır.

Yazması daha kolay kod yazma veya okunması daha kolay kod yazma seçeneği göz önüne alındığında, Python programcıları sizi neredeyse her zaman okunması daha kolay kod yazmaya teşvik edecektir.

Atama Operatörü

Tüm Python komutlarının en önemlisi, aşağıdaki gibi bir atama operatörüdür (assignment statement). Atama operatörü, bir tanımlayıcıdan bir nesneye referans (reference) oluşturur.

In [2]:
```
temperature = 98.6
```

Bu komut, `temperature`'ı bir tanımlayıcı (identifier, **isim** olarak da bilinir) olarak belirler, ve sonra bu tanımlayıcıyı eşittir işaretinin sağ tarafında ifade edilen nesneyle ilişkilendirir. Bu örnek için, `temperature` tanımlayıcısı 98.6 değerine sahip bir kayan-nokta (float) nesnesiyle ilişkilendirilmiştir.

Python'daki tanımlayıcılar (identifiers) büyük/küçük harfe duyarlıdır, bu nedenle `temperature` ve `Temperature` farklı isimlerdir. Tanımlayıcılar, hemen hemen her türlü harf, rakam ve alt çizgi karakterlerinin (veya daha genel Unicode karakterlerinin) birleşiminden oluşabilir. Birincil kısıtlamalar, bir tanımlayıcının bir sayı ile başlayamaması (bu nedenle `9lives` uygun olmayan bir isimlerdir) ve tanımlayıcı olarak kullanılamayan özel olarak ayrılmış (reserved) 33 kelimenin olmasıdır.

Python yorumlayıcısına `help("keywords")` yazarak bu listeyi istediğiniz zaman görebilirsiniz.

In [3]:
```
help("keywords")
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

Bir programcı, mevcut bir nesneye ikinci bir tanımlayıcı atayarak bir takma isim (alias) oluşturabilir.

Python programlamada, bir tanımlayıcının değeri başka bir tanımlayıcıya atandığında, gerçek değeri kopyalamak yerine referansın kopyalandığı **takma isimlendirme (aliasing)** gerçekleşir.

In [ ]:
```
temperature = 98.6
```

In [ ]:
```
original = temperature
```

In [ ]:
```
original
```

Out[ ]:
```
98.6
```

Bir takma isim oluşturulduktan sonra, altta yatan nesneye (underlying object, bu örnek için float nesnesi) erişmek için her iki isim de kullanılabilir.

Her tanımlayıcı, atıfta bulunduğu nesnenin bellek adresiyle (memory address) dolaylı olarak ilişkilidir.

In [5]:
```
id(temperature)
```

Out[5]:
```
4453453520
```

In [6]:
```
id(original)
```

Out[6]:
```
4453453520
```

Görüldüğü üzere, aynı bellek adresinde bu iki isim saklanmaktadır!

Bu durumun detaylarına daha sonra detaylı değinilecektir.

Girintiler

Python, R, C++, Java ve Perl gibi diğer birçok dilde olduğu gibi parantez kullanmak yerine kodu yapılandırmak için beyaz boşluk (whitespace) (sekmeler (tabs) veya boşluklar (spaces)) kullanır. Bir sıralama algoritmasından bir **for** döngüsü düşünün:

```
for x in array: if x < pivot: less.append(x) else: greater.append(x)
```

İki nokta üst üste, girintili bir kod bloğunun başlangıcını belirtir; bundan sonra tüm kod bloğun sonuna kadar aynı miktarda girintili olmalıdır.

Açıklamalar / Yorumlar

Başında kare işareti (hash işareti) `#` olan herhangi bir metin Python yorumcusu tarafından yok sayılır. Bu genellikle koda yorum eklemek için kullanılır. Bazen belirli kod bloklarını silmeden hariç tutmak da isteyebilirsiniz. Kolay bir çözüm, kodu açıklamaya dönüştürmektir:

```
results = [] for line in file_handle: # keep the empty lines for now # if len(line) == 0: # continue results.append(line.replace('foo', 'bar'))
```

In [3]:
```
# print("Hello, World!")
print("Cheers, Mate!")
```

```
#veya

print("Hello, World!")
# print("Cheers, Mate!")
```

```
Cheers, Mate!
Hello, World!
```

Yorumlar, yürütülen bir kod satırından sonra da yazılabilir. Bazı programcılar, belirli bir kod satırından önceki satıra yorumların yerleştirilmesini tercih ederken, bu bazen yararlı olabilir:

In [4]:
```
print("Reached this line") # Simple status report
```

```
Reached this line
```

Python'un çok satırlı (multiline) yorumlar için bir sözdizimi (syntax) yoktur. Çok satırlı bir yorum eklemek için her satıra bir `#` ekleyebilirsiniz:

In [5]:
```
# Bu yorum
# birden çok satıra
# yazılmıştır.
print("Hello, World!")
```

```
Hello, World!
```

Veya çok satırlı bir dizgi (multiline string) kullanabilirsiniz:

In [6]:
```
"""
Bu yorum
birden çok satıra
yazılmıştır.
"""
print("Hello, World!")
```

```
Hello, World!
```

Esasında, çok satırlı dizgiler (multiline strings), uzun bir dizginin farklı satırlara bölünmesine izin veren en basit yöntemdir:

In [7]:
```
line_str = "I'm learning Python.\nMy trainer is named Mustafa Murat Arat.\nHe is th

print("Long string with newlines: \n" + line_str)
```

```
Long string with newlines:
I'm learning Python.
My trainer is named Mustafa Murat Arat.
He is the best teacher for Python programmers. :)))
```

In [8]:
```
multiline_str = """I'm learning Python.
My trainer is named Mustafa Murat Arat.
He is the best teacher for Python programmers. :)))"""

print("Multiline string: \n" + multiline_str)
```

```
Multiline string:
I'm learning Python.
My trainer is named Mustafa Murat Arat.
He is the best teacher for Python programmers. :)))
```

Veri Tipleri

Python, standart kitaplığıyla birlikte sayısal (nümerik) verileri (numerical data), dizgileri (strings), Boole değerleri (Boolean values: `True` (Doğru) veya `False` (Yanlış)) ve tarih ve saati işlemek için küçük bir yerleşik veri tipi grubuna sahiptir. Bu "tek değerli" tiplere bazen skaler tipler (scalar types) denir ve biz bunlara bu eğitimde skaler (scalar) olarak değiniyoruz.

Tip	Tanımı
None	Python "null" değeri
str	Dizgi türü; Unicode (UTF-8 kodlu) dizgilerini tutar
bytes	Ham ASCII baytları (veya bayt olarak kodlanmış Unicode)
float	64 bit kayan noktalı sayı
bool	True (Doğru) veya Flase (Yanlış) bir değer
int	Tamsayı

Nümerik Veri Tipi (Numerical Data Type)

Sayılar için birincil Python türleri `int` ve `float`'tır. Bir `int` keyfi olarak büyük sayıları saklayabilir:

In [9]:
```
ival = 17239871
```

In [10]:
```
ival
```

Out[10]:
```
17239871
```

In [11]:
```
type(ival)
```

Out[11]:
```
int
```

In [12]:
```
ival ** 6
```

Out[12]:
```
26254519291092456596965462913230729701102721
```

Kayan nokta sayılar, Python'da `float` türüyle temsil edilir. Her bir kayan noktalı sayı çift duyarlıklı (double precision) (64 bit) bir değerdir. Bilimsel gösterimle de ifade edilebilirler:

In [13]:
```
fval = 7.243
```

In [14]:
```
fval
```

Out[14]:
```
7.243
```

In [15]:
```
type(fval)
```

Out[15]:
```
float
```

In [16]:
```
# Bilimsel gösterim (scientific notation)

fval2 = 6.78e-5
```

In [17]:
```
fval2
```

Out[17]:
```
6.78e-05
```

In [18]:
```
type(fval2)
```

Out[18]:
```
float
```

Tam sayı ile sonuçlanmayan tamsayı bölümü her zaman bir kayan noktalı sayı verir:

In [19]:
```
3/2
```

Out[19]:
```
1.5
```

In [20]:
```
type(3/2)
```

Out[20]:
```
float
```

C-tarzı tamsayı bölümü elde etmek için (sonuç bir tam sayı değilse kesirli kısmı düşürür), kat bölme operatörünü kullanın `//`:

In [21]:
```
3//2
```

Out[21]:
```
1
```

Dizgi Veri Tipi (String Data Type)

Birçok kişi Python'u güçlü ve esnek yerleşik dizgi işleme yetenekleri için kullanır. Tek tırnak `'` veya çift tırnak `"` kullanarak dizgileri yazabilirsiniz:

In [22]:
```
a = 'one way of writing a string'
a
```

Out[22]:
```
'one way of writing a string'
```

In [23]:
```
type(a)
```

Out[23]:
```
str
```

In [24]:
```
b = "another way"
b
```

Out[24]:
```
'another way'
```

In [25]:
```
type(b)
```

Out[25]:
```
str
```

Tek tırnak `'` veya çift tırnak `"` kullanımının seçimi , özellikle `"Don't worry"` de olduğu gibi bir sekansta gerçek bir karakter olarak başka bir tırnak karakteri kullanıldığında uygundur.

In [14]:
```
c = "Don't worry"
```

In [15]:
```
type(c)
```

Out[15]:
```
str
```

Alternatif olarak, `'Don\'t worry'` de olduğu gibi, bir ters eğik çizgi (backslash) kullanılarak kaçış karakteri olarak alıntı sınırlayıcı kullanılabilir.

In [17]:
```
d = 'Don\'t worry'
```

In [18]:
```
type(d)
```

Out[18]:
```
str
```

Satır kesmeli çok satırlı dizgiler için, `'''` veya `"""` şeklinde üçlü tırnak işaretleri kullanabilirsiniz.

Bu tür üç tırnaklı dizgilerin avantajı, yeni satır karakterlerinin (newline characters), `\n` olarak kaçmak yerine, doğal olarak gömülebilmesidir. Bu, kaynak koddaki uzun, çok satırlı dizgilerin okunabilirliğini büyük ölçüde artırabilir.

In [22]:
```
email = "\nDear Alice,\n\nEve's cat has been arrested for catnapping, cat burglary,
email
```

Out[22]:
```
"\nDear Alice,\n\nEve's cat has been arrested for catnapping, cat burglary, and ex
tortion.\n\nSincerely,\nBob\n"
```

In [21]:
```
email = """
Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob
"""
```

```
email
```

Out[21]:
```
"\nDear Alice,\n\nEve's cat has been arrested for catnapping, cat burglary, and ex
tortion.\n\nSincerely,\nBob\n"
```

In [24]:
```
type(email)
```

Out[24]:
```
str
```

Python dizgileri değiştirilemezdir (**immutable**); bir dizgiyi yarattıktan sonra *değiştiremezsiniz*:

In [28]:
```
a = 'this is a string'
```

In [29]:
```
a[10] = 'f'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_17053/3197216345.py in <m
odule>
----> 1 a[10] = 'f'

TypeError: 'str' object does not support item assignment
```

Bu işlemden sonra, `a` değişkeni değişmemiştir:

In [30]:
```
a
```

Out[30]:
```
'this is a string'
```

Birçok Python nesnesi, `str` fonksiyonu kullanılarak bir dizgiye dönüştürülebilir:

In [25]:
```
a = 5.6
# Bir kayan noktalı sayı
```

In [26]:
```
s = str(a)
s
```

Out[26]:
```
'5.6'
```

In [27]:
```
type(s)
```

Out[27]:
```
str
```

Dizgiler bir Unicode karakter sekansıdır, ve bu nedenle listeler (lists) ve demetler (tuples) gibi diğer **sekans** türleri gibi ele alınabilir (ki bunu bir sonraki bölümde daha ayrıntılı olarak inceleyeceğiz).

Bir sekans (sequence), sıranın önemli olduğu bir değerler koleksiyonunu temsil eder.

In [34]:
```
s = 'python'
```

In [35]:
```
list(s)
#Listede her bir öğe, dizginin bir karakterine denk gelmektedir!
```

Out[35]:
```
['p', 'y', 't', 'h', 'o', 'n']
```

In [36]:
```
for i in s:
    print(i)
```

```
p
y
t
h
o
n
```

In [37]:
```
s[:3]
```

Out[37]:
```
'pyt'
```

`s[:3]` sözdizimi, dilimleme (slicing) olarak adlandırılır ve birçok Python sekans türü için uygulanır.

In [38]:
```
s[0:3] #ile aynı sözdizimi
```

Out[38]:
```
'pyt'
```

Python'un karakterler (characters) için özel bir veri yapısı yoktur. Karakterler 1 uzunluğunda dizgilerdir (string).

In [12]:
```
's'
```

Out[12]:
```
's'
```

In [13]:
```
type('s')
```

Out[13]:
```
str
```

Bir karakterin bir dizgide bulunup bulunmadığını kontrol etmek için `in` deyimi kullanılır:

In [2]:
```
'm' in 'murat'
```

Out[2]:
```
True
```

Boole Veri Tipi (Boolean Data Type)

Python'daki iki boole değeri `True` ve `False` olarak yazılır. Karşılaştırmalar ve diğer koşullu ifadeler, Doğru (True) veya Yanlış (False) olarak değerlendirilir. Boole değerleri `and` ve `or` anahtar sözcükleri ile birleştirilir:

In [39]:
```
True and True
```

Out[39]: True

In [40]:
```
True and False
```

Out[40]: False

In [41]:
```
False and True
```

Out[41]: False

In [42]:
```
False and False
```

Out[42]: False

In [43]:
```
True or True
```

Out[43]: True

In [44]:
```
True or False
```

Out[44]: True

In [45]:
```
False or True
```

Out[45]: True

In [46]:
```
False or False
```

Out[46]: False

Not: Sayılar, sıfırsa `False` olarak ve sıfır değilse `True` olarak değerlendirilir

In [4]:
```
bool(0)
```

Out[4]: False

In [5]:
```
bool(35)
```

Out[5]: True

Not: Sekanslar ve Dizgiler (string) ve listeler (list) gibi diğer konteyner (container) türleri, boşsa `False`, boş değilse `True` olarak değerlendirin.

In [7]:
```
bool('')
```

Out[7]: False

In [8]:
```
bool('murat')
```

Out[8]: True

In [9]:
```
bool([])
```

Out[9]:
```
False
```

In [ ]:
```
bool([3, 6, 9])
```

Out[ ]:
```
True
```

Tip Çevirme (Type Conversion)

`str`, `bool`, `int` ve `float` tipleri de bu tiplere değer atamak için kullanılabilen fonksiyonlardır:

In [47]:
```
s = '3.14159'
```

In [48]:
```
type(s)
```

Out[48]:
```
str
```

In [49]:
```
fval = float(s)
```

In [50]:
```
fval
```

Out[50]:
```
3.14159
```

In [51]:
```
type(fval)
```

Out[51]:
```
float
```

In [52]:
```
int(fval)
```

Out[52]:
```
3
```

In [53]:
```
bool(fval)
```

Out[53]:
```
True
```

In [54]:
```
bool(0)
```

Out[54]:
```
False
```

`str`, `bool`, `int` ve `float` fonksiyonları yerleşik (built-in) Python fonksiyonlarıdır. - https://docs.python.org/3/library/functions.html

None Veri Tipi

`None` Python boş değer (null) veri tipidir. Bir Fonksiyon açıkça bir değer döndürmezse, dolaylı olarak `None` değerini döndürür:

In [55]:
```
a = None
```

In [56]:
```
a is None
```

Out[56]: True

In [57]:
```
b=5
```

In [58]:
```
b is None
```

Out[58]: False

In [59]:
```
b is not None
```

Out[59]: True

`None`, ayrıca bir fonksiyonun argümanları için ortak bir varsayılan değerdir (default):

In [60]:
```
def add_and_maybe_multiply(a, b, c = None):
    result = a + b
    if c is not None:
        result = result * c
    return result
```

In [61]:
```
add_and_maybe_multiply(a = 2, b = 3, c = None)
```

Out[61]: 5

In [62]:
```
add_and_maybe_multiply(a = 2, b = 3, c = 4)
```

Out[62]: 20

İkili operatörler ve karşılaştırmalar

İkili matematik işlemlerinin ve karşılaştırmalarının çoğu beklediğiniz gibidir:

In [63]:
```
5 - 7
```

Out[63]: -2

In [64]:
```
12 + 21.5
```

Out[64]: 33.5

In [65]:
```
5 <= 2
```

Out[65]: False

Mevcut tüm ikili operatörler için aşağıdaki tabloya bakınız:

Operasyon	Tanım
a+b	a ve b'yi toplayın
a-b	a'dan b'yi çıkartın
a*b	a ve b'yi çarpın
a/b	a'yı b'ye bölün
a // b	Herhangi bir kesirli kalanı bırakarak a'yı b'ye bölün
a ** b	a'yı b gücüne yükseltin
a & b	Hem a hem de b True (Doğru) ise True (Doğru)'dur; tamsayılar için bitsel AND ifadesini kullanabilirsiniz.
a | b	a veya b True (Doğru) ise True (Doğru)'dur; tamsayılar için bitsel OR ifadesini kullanabilirsiniz.
a^b	Boole değerleri çin, a veya b True (Doğru) ise True (Doğru), ancak ikisi birden değil; tamsayılar için bit düzeyinde EXCLUSIVE-OR ifadesini kullanabilirsiniz.
a == b	a, b'ye eşit ise True (Doğru)'dur
a != b	a, b'ye eşit değil ise True (Doğru)'dur
a <= b, a < b	a, b'den küçük (veya küçük eşit) ise True (Doğru)'dur
a > b, a >= b	a, b'den büyük (veya büyük eşit) ise True (Doğru)'dur
a is b	a ve b aynı Python nesnesini referans alıyorsa True (Doğru)'dur
a is not b	a ve b farklı Python nesnesini referans alıyorsa True (Doğru)'dur

In [66]:
```
a = 5
b= 7
```

In [67]:
```
a+b
```

Out[67]: 12

In [68]:
```
a-b
```

Out[68]: -2

In [69]:
```
a/b
```

Out[69]: 0.7142857142857143

In [70]:
```
a*b
```

Out[70]: 35

In [71]:
```
a**b
```

Out[71]: 78125

In [72]:
```
b // a # mod alma (modulo operaötrü)
```

Out[72]: 1

In [73]:
```
a == b
```

Out[73]: False

In [74]:
```
a != b
```

Out[74]: True

In [75]:
```
a is b
```

Out[75]: False

In [76]:
```
a is not b
```

Out[76]: True

NOT:

- `is` operatörü referans eşitliği (reference equality) olarak bilinir. İki referansın aynı nesneye referans olup olmadığını (aynı nesneye atıfta bulunup bulunmadığını) bilmek istediğinizde kullanınız.
- `==` operatörü değer eşitliği (value equality) olarak bilinir. İki nesnenin aynı değere sahip olup olmadığını bilmek istediğinizde kullanabilirsiniz.

Peki referans olma ne demektir?

Kodumunuzu yazarken kullanacağımız bir başka önemli fonksiyon ise `id()` 'dir.

In [77]:
```
id(a)
```

Out[77]: 4531739264

In [78]:
```
id(b)
```

Out[78]: 4531739328

`id()` fonksiyonu yerleşik (built-in) Python fonksiyonudur. Gördüğümüz gibi fonksiyon tek bir parametreyi kabul eder ve bir nesnenin kimliğini (id) döndürmek için kullanılır. Bu kimlik, yaşam boyunca (lifetime- yani, kodunuz çalıştığı süre boyunca) bu nesne için benzersiz (unique) ve sabit (fixed) olmalıdır.

Bunu C programlama dili ile ilişkilendirirsek, bu kimlik numaraları aslında bellek (memory) adresidir, burada Python'da benzersiz kimliktir.

Bu fonksiyon genellikle Python'da dahili (internal) olarak kullanılır.

In [79]:
```
# id of 5
print("id of 5 =", id(5))

a = 5

# id of a
print("id of a =", id(a))

b = a

# id of b
print("id of b =", id(b))

c = 5.0

# id of c
print("id of c =", id(c))
```

```
id of 5 = 4531739264
id of a = 4531739264
id of b = 4531739264
id of c = 140297773386448
```

Gördüğünüz gibi, `id()` fonksiyonu hem `a = 5` hem de `5` için 4531739264 tamsayısını döndürür.

Her iki değer de aynı olduğundan, id de aynıdır, çünkü hem `a = 5` hem de `5` arka planda aynı tamsayı (integer) nesnesine atıfta bulunmaktadır:

Esasında `a = 5` demek `5` tam sayı nesnesini `a` isminde sakla demektir.

Ayrıca, `=` operatörü atama (assignment) operatüdür. `b = a` derken `a = 5` tamsayı nesnesinin sığ bir kopyasını (a shallow copy) oluşturur ve `b` ismine atarız. Sığ kopya konseptine daha sonra değinilecektir.

Ancak, `c = 5.0` için farklı bir id dönmüştür, çünkü `c` bir kayan-noktalı (float) nesnesidir. Tamamiyle farklı bir nesnedir.

Fonksiyonlar

Bir fonksiyon (function), yalnızca çağrıldığında çalışan bir kod bloğudur ve belirli bir görevi yerine getiren ilgili ifadeler (expressions) grubudur.

Fonksiyonlar, programımızı daha küçük ve modüler parçalara ayırmaya yardımcı olur. Programınız büyüdükçe, fonksiyonlar programınıza ait kodu daha düzenli ve yönetilebilir hale getirir.

Ayrıca, tekrarı önler ve kodu yeniden kullanılabilir hale getirir.

Parametre (parameter) olarak bilinen verileri bir fonksiyona gönderebilisiniz.

def function_name(parameters): """docstring""" expression(s)

Aşağıdaki örnek, bir fonksiyonun en basit yapısını göstermektedir:

In [92]:
```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")
```

In [93]:
```
greet_user()
```

```
Hello!
```

`print("Hello!")` satırı, bu fonksiyonun gövdesindeki tek gerçek kod satırıdır, bu nedenle `greet_user()` nesnesinin yalnızca bir görevi vardır: `print("Hello!")`.

`def` sözcüğü "definiton" kelimesinden gelmektedir ve fonksiyon tanımlama (function definition) anlamına gelir!

Biraz değiştirilmiş, `greet_user()` fonksiyonu kullanıcıya sadece "Merhaba! (Hello!)" demekle kalmaz, ama aynı zamanda onları isimleriyle selamlamalıdır!

In [94]:
```
def greet_user(username):
    """Display a simple greeting."""
    print(f"Hello, {username}!")
```

In [95]:
```
greet_user('Murat')
```

```
Hello, Murat!
```

`greet_user()` tanımındaki değişken *username*, bir parametre (parameter) örneğidir, fonksiyonun işini yapması için ihtiyaç duyduğu bir bilgi parçasıdır.

`greet_user('Murat')` içindeki `'Murat'` değeri bir argüman (argument) örneğidir. Argüman, bir fonksiyon çağrısından (a function call) bir fonksiyona gönderilen bir bilgi parçasıdır.

Docstring

`greet_user()` fonksiyonunun içerisindeki `"""Display a simple greeting."""` ifadesine docstring (document string) nedir.

Kısaca bir fonksiyonun ne yaptığını açıklamak için kullanılır. İsteğe bağlı olmasına rağmen, dokümantasyon iyi bir programlama uygulamasıdır. Her zaman kodunuzu belgeleyiniz!

Yukarıdaki örnekte, fonksiyon başlığının hemen altında bir *docstring*imiz var. Doküman dizgisinin birden çok satıra (multiline) kadar genişletilebilmesi için genellikle üçlü tırnak

işaretleri kullanırız. Bu dizgi, fonksiyonun `__doc__` niteliği (attribute) olarak bizim için kullanılabilir.

In [96]:
```
greet_user.__doc__ # Dunder Methods (Double Underscore)
```

Out[96]:
```
'Display a simple greeting.'
```

return Deyimi

`return` deyimi, bir fonksiyondan çıkmak ve çağrıldığı yere geri dönmek için kullanılır.

```
return [expression_list]
```

Bu ifade, değerlendirilen ve değer döndürülen bir ifade içerebilir. Deyimde ifade yoksa veya bir işlevin içinde return ifadesinin kendisi yoksa, işlev None nesnesini döndürür.

In [97]:
```
print(greet_user('Murat'))
```

```
Hello, Murat!
None
```

Burada, None döndürülen değerdir, çünkü `greet_user()` doğrudan bir metni ekrana yazdırır ve hiçbir return deyimi kullanılmamıştır.

In [98]:
```
type(greet_user('Murat'))
```

```
Hello, Murat!
```

Out[98]:
```
NoneType
```

Fonksiyon hiçbir şey döndürmediğinden (return), `NoneType` sınıfından (class) bir nesne (object) elde ederiz.

In [99]:
```
def absolute_value(num):
    """This function returns the absolute
    value of the entered number"""

    if num >= 0:
        return num
    else:
        return -num
```

In [100…
```
print(absolute_value(2))
```

```
2
```

In [101…
```
print(absolute_value(-2))
```

```
2
```

`return` deyiminin en güzel tarafı, fonksiyonun dönderdiği değeri bir değişkende saklayabilmenizdir:

In [102…
```
a = greet_user('Murat')
```

```
Hello, Murat!
```

In [103…
```
a
```

Görüldüğü üzere `greet_user` fonksiyonunun `return` deyimi olmadığı için bu fonksiyonun çıktısını `a` gibi bir değişkende sakladığımızda ve a'yı ekrana yazdırdığımızda herhangi bir değer ekrana yazılmamıştır. Aynı şeyi, `absolute_value()` fonksiyonu ile yapalım:

In [104…
```
result_ = absolute_value(2)
result_
```

Out[104…
```
2
```

In [105…
```
result_ = absolute_value(-10)
result_
```

Out[105…
```
10
```

Fonksiyon Argümanları

Aşağıdaki argüman türlerini kullanarak bir fonksiyonu çağırabilirsiniz -

1. Gerekli argümanlar (required arguments)
2. Konumsal argümanlar (positional arguments)
3. Anahtar kelime argümanları (keyword arguments)
4. Varsayılan argümanlar (default arguments)

Gerekli argümanlar (required arguments)

Gerekli argümanlar, bir fonksiyona doğru konum (positional) sırasına göre gönderilen argümanlardır. Burada, fonksiyon çağrısındaki argüman sayısı, fonksiyon tanımıyla (function definition) tam olarak eşleşmelidir.

In [106…
```
def square_(num): #function definition
    return num**2
```

In [107…
```
square_() #function call
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_17053/2547384884.py in <m
odule>
----> 1 square_()

TypeError: square_() missing 1 required positional argument: 'num'
```

Bir sayının karesini döndüren `square_()` fonksiyonunu çağırmak için kesinlikle bir argüman iletmeniz gerekmektedir, aksi takdirde yukarıdaki gibi bir sözdizimi hatası verir.

In [108…
```
square_(num = 2)
```

Out[108…
```
4
```

Konumsal argümanlar (positional arguments)

Python'da konumsal bir argüman, bir fonksiyon çağrısında konumu önemli olan bir argümandır.

```
call_func(arg1, arg2, arg3)
```

İsim ve yaş verilen bir kişi hakkında bilgi gösteren bir fonksiyon tanımlayalım:

In [109…
```
def info(name, age):
    print(f"Hi, my name is {name}. I am {age * 365.25} days old.")
```

Şimdi bu fonksiyonu iki konumsal argüman olan `name` ve `age` ile çağıralım:

In [110…
```
info('Murat', 32)
```

```
Hi, my name is Murat. I am 11688.0 days old.
```

Neden konumsal? Çünkü argümanların sırası önemlidir.

Bu fonksiyonu aynı argümanlarla yanlış sırada çağırırsanız, bir hata alırsınız:

In [111…
```
info(32, 'Murat')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_17053/623821999.py in <mo
dule>
----> 1 info(32, 'Murat')

/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_17053/2304733709.py in in
fo(name, age)
      1 def info(name, age):
----> 2     print(f"Hi, my name is {name}. I am {age * 365.25} days old.")

TypeError: can't multiply sequence by non-int of type 'float'
```

Bu durumda yaşı bir isim (name) ve ismi de yaş (age) olarak ele almaktadır. Bu hatayı almayı engellemek için Anahtar Kelime Argümanları (Keyword Arguments) kullanılır.

Anahtar kelime argümanları (keyword arguments)

Python'da bir anahtar kelime argümanı, bir fonksiyon argümanının bir isim etiketine sahip olduğu anlamına gelir.

```
    call_func(arg_name=arg1)
```

Bir önceki örneğe geri dönelim...

Gördüğünüz gibi, bu fonksiyon iki konumsal argüman olan `name` ve `age` argümanlarını beklemektedir.

Bu, sıranın önemli olduğu anlamına gelir.

Ancak, argümanları anahtar kelime argümanları olarak da iletebilirsiniz.

Bu, fonksiyonu çağırdığınızda argümanları adlandırdığınız anlamına gelir.

In [112…
```
info(age=32, name="Murat")
```

```
Hi, my name is Murat. I am 11688.0 days old.
```

Yukarıdaki fonksiyon çağrısında, argümanların sırasını değiştirdiğime dikkat edin!

Yani, argüman sırası artık önemli değil.

Argüman isimlerini tanımladığımız için Python kolay bir şekilde `age` argümanı için `32` değerini, `name` argümanı için `"Murat"` değerini alacağını anlayacaktır.

Varsayılan argümanlar (default arguments)

Varsayılan argüman, bu argüman için fonksiyon çağrısında (function call) bir değer sağlanmadığında varsayılan bir değer varsayan bir argümandır.

In [113…
```
def info(name, age = 32):
    print(f"Hi, my name is {name}. I am {age * 365.25} days old.")
```

In [114…
```
info(name = 'Ayşe')
```

```
Hi, my name is Ayşe. I am 11688.0 days old.
```

`info(name = 'Ayşe')` fonksiyon çağrısı, `info(name = 'Ayşe', age = 32)` fonksiyon çağrısı ile aynı değeri ekrana yazdırır. Çünkü yaş (`age`) argümanının alacağı değer sabitlenmiştir.

In [115…
```
info(name = 'Ayşe', age = 32)
```

```
Hi, my name is Ayşe. I am 11688.0 days old.
```

Ancak, `age` değerine istediğiniz yeni değeri verebilirsiniz:

In [116…
```
info(name = 'Ayşe', age = 71)
```

```
Hi, my name is Ayşe. I am 25932.75 days old.
```

Anonim Fonksiyonlar (Lambda fonksiyonu)

Bu fonksiyonlar, `def` anahtar sözcüğü kullanılarak standart biçimde deklere edilmedikleri için anonim fonksiyonlar (anonymous functions) olarak adlandırılır.

Küçük anonim fonksiyonlar oluşturmak için `lambda` anahtar sözcüğünü kullanabilirsiniz.

Bu nedenle anonim fonksiyonlara lambda fonksiyonları da denir. Python'daki bir lambda fonksiyonu aşağıdaki sözdizimine (syntax) sahiptir.

```
lambda arguments: expression
```

Lambda fonksiyonları herhangi bir sayıda argümana sahip olabilir, ancak yalnızca bir ifadeye (expression) sahip olabilir. İfade değerlendirilir ve döndürülür. Lambda fonksiyonları, fonksiyon nesnelerinin gerekli olduğu her yerde kullanılabilir.

Örneğin, bir sayının karesini alan bir fonksiyonu önceden yazmıştır:

In [117…
```
def square_(num):
    return num**2
```

In [118…
```
square_(num = 2)
```

Out[118… 4

Bu fonksiyonu bir lambda fonksiyonu olarak kolaylıkla yazabiliriz:

In [1]:
```
square_ = lambda x: x ** 2

print(square_(4))
```

16

Yukarıdaki programda `square_ = lambda x: x * 2`, lambda fonksiyonudur. Burada `x` bir argümandır ve `x * 2`, değerlendirilen ve döndürülen ifadedir.

Yukarıda da dediğimiz gibi Lambda fonksiyonları herhangi bir sayıda argümana sahip olabilir, ancak yalnızca bir ifadeye (expression) sahip olabilir.

In [50]:
```
def iki_sayi_toplami(a, b):
    return a + b

iki_sayi_toplami(a = 8, b = 4)
```

Out[50]: 12

In [53]:
```
iki_sayi_toplami = lambda a, b: a + b
```

In [54]:
```
iki_sayi_toplami(8, 4)
```

Out[54]: 12

Lambda fonksiyonlarının diğer kullanım yerlerini eğitimin ilerleyen bölümlerinde detaylı göreceğiz.

Tabii ki, lambda fonksiyonları da tek bir değer döndermek zorunda değildir:

In [10]:
```
def sum_multiply(a, b):
    return a + b, a * b
```

In [11]:
```
sum_multiply(a = 3, b = 2)
```

Out[11]:
```
(5, 6)
```

In [13]:
```
sum_multiply = lambda a, b: (a + b, a * b)
sum_multiply(a = 3, b = 2)
```

Out[13]:
```
(5, 6)
```

Birinci-sınıf fonksiyon (first-class function)

Birinci-sınıf bir fonksiyon (first-class function), belirli bir fonksiyon türü değildir. Python'daki tüm fonksiyon birinci sınıf fonksiyonlardır. Belirli bir programlama dilinde fonksiyonların birinci sınıf olduğunu söylemek, diğer nesne türleri (tamsayılar veya dizgiler gibi) gibi kullanılabileceği ve manipüle edilebileceği anlamına gelir. Yani, Python'da bir fonksiyon:

- Fonksiyonlar da birer nesnedir ve değişkenlere atanabilir.
- Bir döndürülen değer (return value) olarak kullanılabilir
- Parametre olarak kullanılabilir.
- Hash tabloları, listeler gibi veri yapılarında saklanabilir...

Fonksiyonlar da birer nesnedir ve değişkenlere atanabilir.

In [1]:
```
def yell(text):
    return text.upper() + '!'
```

In [2]:
```
yell('hello')
```

Out[2]:
```
'HELLO!'
```

In [3]:
```
bark = yell
```

In [4]:
```
bark('woof')
```

Out[4]:
```
'WOOF!'
```

In [5]:
```
bark.__name__
```

Out[5]:
```
'yell'
```

Bir fonksiyon nesnesi ve bir fonksiyonun ismi (name) iki ayrı konudur. Bir fonksiyonun orijinal ismini (burada, `yell` fonksiyonu) silebilirsiniz. Başka bir isim (burada, `bark` fonksiyonu) hala temel fonksiyona işaret ettiğinden, fonksiyonu yine de onun aracılığıyla çağırabilirsiniz:

In [6]:
```
del yell
```

In [7]:
```
yell('hello?')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_5086/494358236.py in <mod
ule>
----> 1 yell('hello?')

NameError: name 'yell' is not defined
```

In [8]:
```
bark('hello')
```

Out[8]:
```
'HELLO!'
```

In [9]:
```
bark.__name__
```

Out[9]:
```
'yell'
```

Fonksiyonlar bir döndürülen değer (return value) olarak kullanılabilir.

Bu opsiyon çoğunlukla özyineleme (recursion) dediğim bir konsept için kullanılır.

Özyineleme, bir şeyi kendi içinde tanımlama sürecidir.

Fiziksel bir dünya örneği, birbirine bakan iki paralel ayna yerleştirmek olacaktır. Aralarındaki herhangi bir nesne özyinelemeli olarak yansıtılacaktır.

In [14]:
```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


num = 3
print("The factorial of", num, "is", factorial(num))
```

```
The factorial of 3 is 6
```

Fonksiyonlar Diğer Fonksiyonlara Gönderilebilir

Fonksiyonlar da nesne (object); olduğundan, bunları diğer fonksiyonlara argüman olarak iletebilirsiniz.

In [15]:
```
def yell(text):
    return text.upper() + '!'
```

In [16]:
```
def greet(func):
    greeting = func('Hi, I am a Python program')
    print(greeting)
```

In [17]:
```
greet(yell)
```

```
HI, I AM A PYTHON PROGRAM!
```

Fonksiyonlar Veri Yapılarında Depolanabilir

Fonksiyonlar birinci sınıf vatandaşlar olduğundan, onları diğer nesnelerde olduğu gibi veri yapılarında saklayabilirsiniz. Örneğin, bir listeye işlevler ekleyebilirsiniz:

In [18]:
```
def yell(text):
    return text.upper() + '!'

bark = yell
```

In [19]:
```
funcs = [bark, str.lower, str.capitalize]
```

In [20]:
```
funcs
```

Out[20]:
```
[<function __main__.yell(text)>,
 <method 'lower' of 'str' objects>,
 <method 'capitalize' of 'str' objects>]
```

In [22]:
```
for f in funcs:
    print(f('hey there'))
```

```
HEY THERE!
hey there
Hey there
```

Ayrıca, indeksleme ile bu listedeki herhangi bir fonksiyonu çağırıp çarıştırabilirsiniz:

In [23]:
```
funcs[0]('heyho')
```

Out[23]:
```
'HEYHO!'
```

`func` ve `func()` yazım farkı

Yukarıda da bahsettiğimiz gibi Python ayrıca fonksiyonları dönüş değerleri (return values) olarak kullanmanıza izin verir.

Aşağıdaki örnek, dış `parent()` fonksiyonundan iç fonksiyonlardan birini döndürür:

In [2]:
```
def parent(num):
    def first_child():
        return "Hi, I am Emma"

    def second_child():
        return "Call me Liam"

    if num == 1:
        return first_child
    else:
        return second_child
```

Parantezler olmadan `first_child` ve `second_child` döndürdüğünüze dikkat ediniz!

In [3]:
```
first = parent(1)
```

In [4]:
```
first
```

Out[4]:
```
<function __main__.parent.<locals>.first_child()>
```

In [5]:
```
second = parent(2)
```

In [6]:
```
second
```

Out[6]:
```
<function __main__.parent.<locals>.second_child()>
```

`first` değişkenin `parent()` içindeki yerel `first_child()` fonksiyonuna refere ettiği, `second` değişkeninin ise `second_child()` fonksiyonuna refere ettiği anlamına gelir.

Buna karşılık parantezli `first_child()` fonksiyonu bir değerlendirmenin (evaluation) sonucunu ifade eder. Aynı durum, `second_child()` fonksiyonu için de geçerlidir!

Refere ettikleri fonksiyonlara doğrudan erişilemese de, artık `first` ve `second` isimlerini normal fonksiyonlarmış gibi kullanabilirsiniz:

In [8]:
```
first()
```

Out[8]:
```
'Hi, I am Emma'
```

In [9]:
```
second()
```

Out[9]:
```
'Call me Liam'
```

Değişkenlerin Kapsamı (Scope of Variables)

Bir programdaki tüm değişkenlere o programdaki tüm konumlardan (locations) erişilemeyebilir. Bu, bir değişkeni nerede bildirdiğinize bağlıdır.

Programlama dillerinde değişkenler kullanılmadan önce tanımlanmalıdır (defined). Bu değişkenlere yalnızca tanımlandıkları alanda erişilebilir, buna kapsam (scope) denir. Bunu değişkenlere erişebileceğiniz bir blok olarak düşünebilirsiniz.

Python'da iki temel değişken kapsamı vardır -

1. Global Kapsam (Global Scope)
2. Yerel Kapsam (Local Scope)
3. Kapsayıcı Kapsam (Enclosing Scope)
4. Yerleşik Kapsam (Built-in)

Bir fonksiyon gövdesi (yani, kod bloğu) içerisinde tanımlanan değişkenlerin yerel bir kapsamı, dışında tanımlananların ise global bir kapsamı vardır.

Yerel Kapsam (Local Scope)

Yerel kapsam değişkenlerine yalnızca kendi bloğu içinde erişilebilir. Bu kapsama, fonksiyon kapsamı (function scope) da denir!

Bir örnekle görelim.

In [127…

```
num1 = 10 #global değişken
def function():
    print("Hello")
    num2 = 20 # yerel değişken çünkü fonksiyonun kod bloğunda

function()
print(num1)
print(num2)
```

```
Hello
10
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_17053/1895750408.py in <m
odule>
      6 function()
      7 print(num1)
----> 8 print(num2)

NameError: name 'num2' is not defined
```

Yukarıdaki örnekte Python'un `num1` değişkeninin değerini yazdırdığını ancak `num2` değişkenini bulamadığını görüyoruz. Bunun nedeni, `num2` 'nin fonksiyonda bir yerel kapsamda (local scope) tanımlanmış olmasıdır, bu nedenle, fonksiyon dışında bu yerel değişkene erişemiyoruz. Bu, yerel kapsamın doğasıdır.

Global Kapsam (Global Scope)

Global kapsama, modül kapsamı (module scope) da denir.

Global kapsamda deklere edilen değişkenlere programın herhangi bir yerinden erişilebilir. Global değişkenler herhangi bir fonksiyonun içinde kullanılabilir. Global değişken değerini de değiştirebiliriz.

In [121…
```
Msg = "Bu mesaj global olarak deklere edilmiştir!"

def function():
    #fonksiyonun yerel kapsamı
    print(Msg)

function()
```

```
Bu mesaj global olarak deklere edilmiştir!
```

Ancak, bir fonksiyon içinde global bir değişkenle aynı isme (name) sahip yerel bir değişken deklere ederseniz ne olur?

In [123…
```
Msg = "Bu mesaj global olarak deklere edilmiştir!"

def function():
    Msg = "Bu mesaj yerel olarak deklere edilmiştir!"
    print(Msg)

function()
print(Msg)
```

```
Bu mesaj yerel olarak deklere edilmiştir!
Bu mesaj global olarak deklere edilmiştir!
```

Gördüğünüz gibi, global değişkenle aynı isme sahip bir yerel değişken deklere edersek, yerel kapsam (local scope) yerel değişkeni (local variable) kullanır.

Global değişkeni yerel kapsamda kullanmak istiyorsanız, bu eğitimin ilerleyen bölümlerinde tartışacağımız "global" anahtar kelimesini kullanmanız gerekecektir.

Kapsayıcı Kapsam (Enclosing Scope)

Yerel veya global olmayan bir kapsam, kapsayıcı kapsamına (enclosing scope) girer. Kapsayıcı kapsama (enclosing scope), yerel-olmayan kapsam (non-local scope) da denir! Bu kapsam, özel bir kapsamdır ve yalnızda iç içe geçmiş fonksiyonlar (nested functions) için mevcuttur.

```
def f():
    def g():
        .
         .
          .
```

Burada `g()` fonksiyonu, `f()` fonksiyonunun içerisindedir.

Yerel kapsam bir iç veya iç içe geçmiş fonksiyonsa (yani, `g()` fonksiyonun kapsamı), bu fonksiyonu çevreleyen kapsam (yani, `f()` fonksiyonun kapsamı), dış veya kapsayıcı fonksiyonun kapsamıdır.

Bu kapsam, çevreleyen fonksiyonda (yani yukarıdaki örnekte `f()` fonksiyonu) tanımladığınız isimleri (names) içerir. Kapsayıcı kapsamdaki isimler, iç ve çevreleyen fonksiyonun kodundan erişilebilir.

In [12]:
```
Msg = "Bu mesaj global olarak deklere edilmiştir!"

def outer_func():
    Msg = "Bu mesaj yerel olmadan deklere edilmiştir!"
    def inner_func():
        print(Msg)
    inner_func()
```

In [14]:
```
outer_func()
```

```
Bu mesaj yerel olmadan deklere edilmiştir!
```

Yerleşik Kapsam (Built-in Scope)

Yerleşik kapsam, Python'daki en geniş kapsamdır. Python yerleşik modüllerindeki tüm ayrılmış (reserved) isimlerin yerleşik bir kapsamı vardır.

Python yerel, kapsayıcı veya global kapsamlarda bir isim (name) bulamadığında, orada tanımlanıp tanımlanmadığını görmek için yerleşik kapsama bakar.

Aşağıdaki kodda görebileceğiniz gibi, yerleşiklerdeki isimlerin her zaman global Python kapsamınıza `__builtins__` özel ismiyle yüklendiğine dikkat ediniz.

`__builtins__` 'nin kendisini `dir()` fonksiyonunu kullanarak incelerseniz, Python yerleşik isimlerinin (built-in names) tam listesini alırsınız.

In [4]:
```
dir(__builtins__)
```

Out[4]:
```
['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BaseException',
 'BlockingIOError',
 'BrokenPipeError',
 'BufferError',
 'BytesWarning',
 'ChildProcessError',
 'ConnectionAbortedError',
 'ConnectionError',
 'ConnectionRefusedError',
 'ConnectionResetError',
 'DeprecationWarning',
 'EOFError',
 'Ellipsis',
 'EnvironmentError',
 'Exception',
 'False',
 'FileExistsError',
 'FileNotFoundError',
 'FloatingPointError',
 'FutureWarning',
 'GeneratorExit',
 'IOError',
 'ImportError',
 'ImportWarning',
 'IndentationError',
 'IndexError',
 'InterruptedError',
 'IsADirectoryError',
 'KeyError',
 'KeyboardInterrupt',
 'LookupError',
 'MemoryError',
 'ModuleNotFoundError',
 'NameError',
 'None',
 'NotADirectoryError',
 'NotImplemented',
 'NotImplementedError',
 'OSError',
 'OverflowError',
 'PendingDeprecationWarning',
 'PermissionError',
 'ProcessLookupError',
 'RecursionError',
 'ReferenceError',
 'ResourceWarning',
 'RuntimeError',
 'RuntimeWarning',
 'StopAsyncIteration',
 'StopIteration',
 'SyntaxError',
 'SyntaxWarning',
 'SystemError',
```

```
'SystemExit',
'TabError',
'TimeoutError',
'True',
'TypeError',
'UnboundLocalError',
'UnicodeDecodeError',
'UnicodeEncodeError',
'UnicodeError',
'UnicodeTranslateError',
'UnicodeWarning',
'UserWarning',
'ValueError',
'Warning',
'ZeroDivisionError',
'__IPYTHON__',
'__build_class__',
'__debug__',
'__doc__',
'__import__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'abs',
'all',
'any',
'ascii',
'bin',
'bool',
'breakpoint',
'bytearray',
'bytes',
'callable',
'chr',
'classmethod',
'compile',
'complex',
'copyright',
'credits',
'delattr',
'dict',
'dir',
'display',
'divmod',
'enumerate',
'eval',
'exec',
'execfile',
'filter',
'float',
'format',
'frozenset',
'get_ipython',
'getattr',
'globals',
```

```
 'hasattr',
 'hash',
 'help',
 'hex',
 'id',
 'input',
 'int',
 'isinstance',
 'issubclass',
 'iter',
 'len',
 'license',
 'list',
 'locals',
 'map',
 'max',
 'memoryview',
 'min',
 'next',
 'object',
 'oct',
 'open',
 'ord',
 'pow',
 'print',
 'property',
 'range',
 'repr',
 'reversed',
 'round',
 'runfile',
 'set',
 'setattr',
 'slice',
 'sorted',
 'staticmethod',
 'str',
 'sum',
 'super',
 'tuple',
 'type',
 'vars',
 'zip']
```

Yerleşik kapsam, mevcut global Python kapsamınıza 150'den fazla isim getirir:

```
In [5]: len(dir(__builtins__))
```

```
Out[5]: 155
```

Tüm bu Python yerleşik nesnelerine ücretsiz olarak (hiçbir şeyi içe aktarmadan (`import`)) erişebilmenize rağmen, `builtins` modülünü açıkça içe aktarabilir ve nokta gösterimini kullanarak adlara erişebilirsiniz. (https://stackoverflow.com/a/76643370/1757224)

In [7]:
```
a = 5.2
type(a)
```

Out[7]:
```
float
```

In [8]:
```
import builtins

builtins.type(a)
```

Out[8]:
```
float
```

Yukarıdaki iki kod satırı aynıdır!

Diğer bir örnek:

In [2]:
```
# without explicitly calling
# the builtins module
print(round(3.14))
```

```
3
```

In [3]:
```
# explicitly calling the
# builtins module
import builtins

a = builtins.round(3.14)
print(a)
```

```
3
```

Böylelikle yerleşik Python fonksiyonlarına bu şekilde de erişebilirsiniz.

En iyi uygulama olarak, hiç bir şekilde bir değişken ismi için python yerleşik isimlerinden birini kullanmamanız gerekmektedir.

Genel isimlerinizden herhangi biri herhangi bir yerleşik ismi geçersiz kılarsa, bir ismi çakışması (name collision) olmayacağından emin olmak istiyorsanız bu oldukça yararlı olabilir.

In [10]:
```
num = -10

#abs bir Python yerleşik ismidir ve builtins modülünde bulunur
abs(num) # callable objects - çağrılabilir nesneler
```

Out[10]:
```
10
```

In [13]:
```
abs = 20 # global kapsamda bir değişken oluşturur

abs(num)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_20201/1021994389.py in <m
odule>
      1 abs = 20
      2
----> 3 abs(num)

TypeError: 'int' object is not callable
```

`abs` yerleşik fonksiyonunu geçersiz kılar (override) veya yeniden atarsanız (reassign), orijinal yerleşik `abs()`, kodunuzun her yerinde etkilenir.

Şimdi, orijinal `abs()` fonksiyonunu çağırmanız gerektiğini ve ismi yeniden atadığınızı unuttuğunuzu varsayalım. Bu durumda, `abs()`'i tekrar çağırdığınızda, bir `TypeError` alırsınız çünkü `abs` artık çağrılabilir (callable) olmayan bir tam sayıya (integer) referans tutar.

Peki, bu problemi nasıl düzeltiriz?

global Python kapsamınızdan yeniden tanımlamayı kaldırmak için oturumunuzu (session, veya çekirdeği (kernel)) yeniden başlatabilir veya `del` ismini çalıştırabilirsiniz. Bu şekilde, yerleşik kapsamdaki orijinal adı geri yüklersiniz. `abs()` örneğine bakacak olursak:

In [14]:
```
del abs

abs(num)
```

Out[14]: 10

Özelleştirilmiş `abs` ismini sildiğinizde, bu ismi global kapsamınızdan kaldırmış olursunuz. Bu, yerleşik kapsamdaki (builtin scope) orijinal `abs()` fonksiyonuna yeniden erişmenizi sağlar.

Bu tür bir duruma geçici bir çözüm bulmak için, yukarıda bahsedildiği gibi `builtins` modülünü açıkça içe aktarabilir (import) ve ardından aşağıdaki kod parçasında olduğu gibi tam nitelikli isimleri kullanabilirsiniz:

In [15]:
```
import builtins

num = -10
builtins.abs(num)
```

Out[15]: 10

`builtins` modülünü açıkça içe aktardıktan sonra, global Python kapsamınızda modül ismine sahip olursunuz.

LEGB Kuralı

https://www.programiz.com/python-programming/namespace

Bu, Python literatüründe yaygın olarak adlandırıldığı şekliyle **LEGB kuralı**dır (Local, Enclosing, Global, Built-in).

LEGB kuralı, Python'un adları arama sırasını belirleyen bir tür isim arama prosedürüdür. Yorumlayıcı (interpreter), yerel, kapsayıcı, global ve son olarak yerleşik kapsamda bakarak bir ismi içten dışa arar:

Kodunuz `x` gibi bir isime refere ediyorsa, Python aşağıdaki isim alanlarında (namespace) gösterilen sırayla `x`'i arar:

1. **Yerel**: Bir fonksiyonun içinde x'i refere ediyorsanız, yorumlayıcı önce onu o fonksiyon için yerel olan en içteki kapsamda arar.
2. **Kapsayıcı**: x yerel kapsamda (local scope) değilse ancak başka bir fonksiyonun içinde bulunan bir fonksiyonda görünüyorsa, yorumlayıcı kapsayıcı fonksiyonun kapsamını arar.
3. **Global**: Yukarıdaki aramaların hiçbiri verimli olmazsa, yorumlayıcı bir sonraki adımda global kapsama bakar.
4. **Yerleşik**: x'i başka hiçbir yerde bulamazsa, yorumlayıcı yerleşik kapsamı dener.

Yorumlayıcı, ismi (name) bu konumların (locations) hiçbirinde bulamazsa, Python bir `NameError` istisnası (Exception) oluşturur.

Örnek 1

In [128…
```
x = 'global'

def f():
    def g():
        print(x)
    g()
```

In [129…
```
f()
```

```
global
```

İlk örnekte, x yalnızca bir konumda tanımlanmıştır. Hem `f()` hem de `g()` dışındadır, dolayısıyla global kapsamda bulunur.

Örnek 2

In [135…
```
x = 'global'

def f():
    x = 'enclosing'
    def g():
        x = 'local'
        print(x)
    g()
```

In [136…
```
f()
```

```
local
```

Bu örnekte, `x` 'in tanımı, `g()` 'nin içerisindedir ve `g()` , x'i kendi içerisinde bulur.

LEGB kuralına göre Python önce yerel (local) kapsama bakar, ve `x` 'i `g()` 'nin içerisinde bulur ve yazdırır!

Örnek 3

In [137…
```
x = 'global'

def f():
    x = 'enclosing'
    def g():
        print(x)
    g()
```

In [138…
```
f()
```

```
enclosing
```

Bu örnekte, `x` 'in tanımı, biri `f()` 'nin dışında, diğeri `f()` 'nin içinde, ancak `g()` 'nin dışında olmak üzere iki yerde görünmektedir.

LEGB kuralına göre Python önce yerel (local) kapsama bakar, yani `g()` 'nin içerisine. Ancak, `x` burada tanımlanmamıştır. Daha sonra, kapsayıcı (enclosing) kapsama bakar. `x` 'i burada bulur!

Örnek 4

Son olarak, `g()` 'nin `x` 'in değerini yazdırmaya çalıştığı ancak `x` 'in hiçbir yerde tanımlanmadığı bir durumumuz var. Bu hiç işe yaramayacak:

In [1]:
```
def f():
    def g():
        print(x)
    g()
```

In [2]:
```
f()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_17709/3782956317.py in <m
odule>
----> 1 f()

/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_17709/2556872193.py in f
()
      2     def g():
      3         print(x)
----> 4     g()

/var/folders/z_/r0991b310gjd66vy5k26g4800000gn/T/ipykernel_17709/2556872193.py in g
()
      1 def f():
      2     def g():
----> 3         print(x)
      4     g()

NameError: name 'x' is not defined
```

Bu sefer Python, isim alanlarının hiçbirinde `x` bulamaz, bu nedenle 3. satırdaki `print()` ifadesi bir `NameError` istisnası oluşturur

Kontrol Akışı (Control Flow)

Bir programın kontrol akışı, program kodunun yürütüldüğü (execution) sıradır (order).

Bir Python programının kontrol akışı, koşullu ifadeler (conditional statements), döngüler (loops) ve fonksiyon çağrıları (function calls) tarafından düzenlenir.

Python, koşullu mantık (conditional logic), döngüler (loops) ve diğer programlama dillerinde bulunan diğer standart kontrol akışı kavramları için birkaç yerleşik anahtar kelimelere sahiptir.

if, elif ve else yapısı

if ifadesi, en iyi bilinen kontrol akışı ifadesi türlerinden biridir. `True` ise devam eden bloktaki kodu değerlendiren bir koşulu kontrol eder.

Bir koşullu yapıda her koşul bir boole ifadesidir! Yani, o koşul `True` olarak değerlendirildiğinde, koşula ait kod bloğu çalıştırılır.

Basit if

if ifadesi, yalnızca belirli bir koşul karşılandığında veya yerine getirildiğinde belirli bir kodu çalıştırmamıza yardımcı olan kontrol akışı ifadeleridir. Basit bir if ifadesinin kontrol edilecek yalnızca bir koşulu vardır.

In [16]:
```
n = 10

if n % 2 == 0:
    print("n is an even number")
```

```
n is an even number
```

In [17]:
```
n = 11

if n % 2 == 0:
    print("n is an even number")
```

Hiç bir çıktı elde edilmez çünkü if yapısı sadece çift (even) sayılar için tanımlanmıştır. Peki ne yapabiliriz?

if-else

if-else ifadesi koşulu değerlendirir ve koşul `True` ise if öğesinin gövdesini yürütür, ancak koşul False ise else öğesinin gövdesi yürütülür.

```
if test expression:
    Body of if
else:
    Body of else
```

In [22]:
```
n = 10

if n % 2 == 0:
    print("n çift sayıdır.")
else:
    print("n tek sayıdır.")
```

```
n çift sayıdır.
```

In [21]:
```
n = 11

if n % 2 == 0:
    print("n çift sayıdır.")
else:
    print("n tek sayıdır.")
```

```
n tek sayıdır.
```

İç içe geçmiş (nested)) if

İç içe if ifadeleri, başka bir if ifadesinin içerisindeki bir if ifadesidir.

In [24]:
```
a = 5
b = 10
```

```
c = 15

if a > b:
    if a > c:
        print("a değeri büyüktür.")
    else:
        print("c değeri büyüktür.")
elif b > c:
    print("b değeri büyüktür.")
else:
    print("c büyüktür.")
```

```
c büyüktür.
```

if-elif-else

if-elif-else ifadesi, bir ifade veya bir ifadeler bloğunu koşullu olarak yürütmek için kullanılır.

```
if test expression:
    Body of if
elif test expression:
    Body of elif
else:
    Body of else
```

In [25]:
```
x = 15
y = 12

if x == y:
    print("Her iki sayı birbirine eşittir.")
elif x > y:
    print("x, y'den büyüktür.")
else:
    print("x, y'den küçüktür.")
```

```
x, y'den büyüktür.
```

NOT: Koşullu ifadelerde, koşullardan herhangi biri `True` ise, aşağıda takip eden herhangi bir başka elif veya else bloklarına ulaşılmayacaktır.

Üçlü operatör (Ternary operator)

Üçlü operatör, Python'da koşullu ifadeler yazmanın bir yoludur. Üçlü adından da anlaşılacağı gibi, bu Python operatörü üç operatörden oluşur.

1. Condition (Koşul): `True` veya `False` olarak değerlendirilen bir boole ifadesi.
2. true_val: Koşul `True` olarak değerlendirilirse atanacak bir değer.
3. false_val: Koşul `Frue` olarak değerlendirilirse atanacak bir değer.

```
var = true_val if condition else false_val
```

Bu üçlü operatörün, if-else yapısı karşılığı şu şekildedir:

```
if condition:
    true_val
else:
    false_val
```

In [34]:
```
n = 10

if n % 2 == 0:
    print("n çift sayıdır.")
else:
    print("n tek sayıdır.")
```

```
n çift sayıdır.
```

In [35]:
```
n = 10

if n % 2 == 0:
    print("n çift sayıdır.")
else:
    print("n tek sayıdır.")
```

```
n çift sayıdır.
```

In [36]:
```
n = 10

result = "n çift sayıdır." if n % 2 == 0 else "n tek sayıdır."
```

In [37]:
```
result
```

Out[37]:
```
'n çift sayıdır.'
```

for döngüleri

for döngüleri, bir koleksiyon (bir liste (list), demet (tuple), sözlük (dictionary) veya küme (set) gibi) veya bir yineleyici (iterator) üzerinde yineleme (iteration) yapmak içindir. Bir for döngüsü için standart sözdizimi şöyledir:

```
for value in collection:
    # value değerini al ve bu değer ile bir şeyler yap
```

In [26]:
```
# sequence isimli listedeki her bir öğeyi döndürecek ve ekrana yazdıracak!
sequence = [1, 2, 4, 5]

for i in sequence:
    print(i)
```

```
1
2
4
5
```

break ifadesi

break ifadesi, onu içeren döngüyü sonlandırır. Programın kontrolü, döngünün gövdesinden hemen sonraki ifadeye akar.

break ifadesi iç içe bir döngü içindeyse (yani, döngü başka bir döngü içindeyse - nested loop), break ifadesi en içteki döngüyü sonlandıracaktır.

```
for i in sequence1:
    for j in sequence 2:
        .
         .
          .
```

In [29]:
```
# Use of break statement inside the loop

sequence = [1, 2, 4, 5, 8, 16, 31, 7, 6]

for val in sequence: #aşağıdaki 3 satır kod, for döngüsünün kod bloğudur (döngünün
    if val == 8:
        break
    print(val)

print("The end") # döngü gövdesinden hemen sonraki ifade
```

```
1
2
4
5
The end
```

continue ifadesi

continue ifadesi, yalnızca geçerli yineleme için bir döngü içindeki kodun geri kalanını atlamak için kullanılır. Döngü sonlandırılmaz, ancak bir sonraki yineleme (iterasyon, iteration) ile devam eder.

In [30]:
```
# Use of break statement inside the loop

sequence = [1, 2, 4, 5, 8, 16, 31, 7, 6]

for val in sequence: #aşağıdaki 3 satır kod, for döngüsünün kod bloğudur (döngünün
    if val == 8:
        continue
    print(val)

print("The end") # döngü gövdesinden hemen sonraki ifade
```

```
1
2
4
5
16
31
7
6
The end
```

Bu program, break ifadesinin continue ifadesi ile değiştirilmesi dışında yukarıdaki örnekle aynıdır.

Görüldüğü üzere 8 değeri ekrana yazılmadı!

pass ifadesi

for döngülerinde kullanılan diğer bir ifade ise pass ifadesidir. null anlamına gelmektedir.

Diyelim ki henüz implement etmediğiniz bir döngü veya fonksiyonumuz var, ancak bu kodu gelecekte yazmayı planlıyorsunuz. Boş bir gövdeye sahip olamazlar. Yorumlayıcı hata verecektir. Bu nedenle, hiçbir şey yapmayan bir gövde oluşturmak için pass ifadesini kullanırız.

pass ifadesi, daha sonra eklenecek bir işlevsellik için yalnızca bir yer tutucudur (placeholder).

In [31]:
```
sequence = {'p', 'a', 's', 's'}
#TODO
for val in sequence:
    pass
```

Aynı şeyi boş bir fonksiyon (function) veya sınıfta (class) da yapabiliriz.

In [32]:
```
#TODO
def function(args):
    pass
```

In [33]:
```
#TODO
class Example:
    pass
```

print fonksiyonu

`print()` fonksiyonu, belirtilen mesajı ekrana veya diğer standart çıktı aygıtına yazdırır.

Mesaj bir dizgi (string) veya başka bir nesne olabilir, nesne ekrana yazılmadan önce bir dizgiye (string) dönüştürülür.

In [39]:
```
# bir yeni satır (newline) yazdırır
print()
```

In [40]:
```
print("Hello", "how are you?")
#print("Hello", "how are you?", sep=' ')
```

```
Hello how are you?
```

In [41]:
```
x = ("apple", "banana", "cherry") # a tuple of strings
print(x)
```

```
('apple', 'banana', 'cherry')
```

In [42]:
```
print("Hello", "how are you?", sep="---")
```

```
Hello---how are you?
```

`+` operatörü dizgileri (string) birleştirmek için kullanılır. `print` fonksiyonunu kullanarak birleştirilmiş dizgiyi ekrana yazdırabilirsiniz:

In [6]:
```
print("merhaba " + "benim adım " + "murat")
```

```
merhaba benim adım murat
```

Bunun kullanım amacı daha çok kod çalışması sırasında güncellenen değişkenleri ekrana yazdırmaktır:

In [7]:
```
x = 5
print("sonuç değeri: " + str(x))
```

```
sonuç değeri: 5
```

Not: `str()` fonksiyonu, bir nesneyi dizgi (string) temsiline dönüştüren global bir yerleşik (built-in) fonksiyondur. (https://docs.python.org/3/library/functions.html)

In [8]:
```
x = 5
type(x)
```

Out[8]:
```
int
```

In [9]:
```
str(x)
```

Out[9]:
```
'5'
```

In [10]:
```
type(str(x))
```

Out[10]:
```
str
```

f-dizgiler (f-strings)

Python 3.6'dan itibaren, f-string'ler, stringleri biçimlendirmenin yeni ve harika bir yoludur. Diğer biçimlendirme yollarından daha okunaklı, daha kısa ve hataya daha az eğilimli olmakla kalmazlar, aynı zamanda daha hızlıdırlar!

f-strings, Formatted string literals demektir.

f-dizgiler ile ilgili her şeyi, Eric V. Smith tarafından Ağustos 2015'te yazılan PEP 498'de okuyabilirsiniz - https://peps.python.org/pep-0498/

f-dizgileri, f sözcüğü ile başlar. Dizgi içerisinde, değerleriyle değiştirilecek ifadeler küme parantezleri içerisinde verilir.

In [44]:
```
name = "Eric"
age = 74

f"Hello, {name}. You are {age}."
```

Out[44]:
```
'Hello, Eric. You are 74.'
```

Büyük F harfi kullanmak da geçerli olacaktır:

In [45]:
```
F"Hello, {name}. You are {age}."
```

Out[45]:
```
'Hello, Eric. You are 74.'
```

Oluşturduğunuz bu dizgiyi `print` fonksiyonuna sarmalayarak ekrana mesajlarınızı veya değişkenlerinizin değerlerini yazdırabilirsiniz:

In [48]:
```
name = "Eric"
age = 74

print(f"Hello, {name}. You are {age}.")
```

```
Hello, Eric. You are 74.
```

f-dizgilerinin içerisine herhangi bir Python programı da yazabilirsiniz:

In [49]:
```
birinci_rakam = 5
ikinci_rakam = 3

print(f'Rakamların toplamı {birinci_rakam + ikinci_rakam} eder.')
```

```
Rakamların toplamı 8 eder.
```

In [1]:
```

```

```
2020 Toyota Corolla with 15000 miles
You've driven 500 miles. The car's new mileage is 15500 miles.
```

In [2]:
```

```

```
2018 Honda Civic with 30000 miles
```

In [ ]:
```

```